sun microsystems

NeWS™ 2.0 Programmer's Guide

Part No: 800-2379-10
Revision A, of 25 August 1989

Contents

Tables

Figures

Preface

This manual provides a guide to programming in the NeWS™ language. This language is supported as part of the X11/NeWS™ server, which itself forms a part of the OpenWindows™ distributed window system.

The NeWS interpreted programming language is based on the POSTSCRIPT® language.[1] Developed at Adobe Systems, the POSTSCRIPT language is a general programming language used primarily for specifying the visual appearance of printed documents. The NeWS language uses POSTSCRIPT language operators to display text and images on a graphics console. Importantly, the NeWS language also provides operators and types that are extensions to the POSTSCRIPT language; many of these extensions handle the interactive aspects of window management that the POSTSCRIPT language does not consider.

This manual, which assumes the reader's familiarity with the POSTSCRIPT language, describes all the basic concepts of NeWS programming. It also provides a syntactic analysis for each NeWS operator and includes code examples that demonstrate the use of NeWS operator and type extensions.

For information about using the X11/NeWS server, see:

- *X11/NeWS Server Guide*
- *X11/NeWS Release Notes*

For information about OpenWindows, see:

- *OpenWindows User's Guide*
- *OpenWindows Installation and Startup Guide*

For information about the POSTSCRIPT language, see:

- *POSTSCRIPT Language Tutorial and Cookbook*[2]
- *POSTSCRIPT Language Reference Manual*[3]

[1] POSTSCRIPT is a registered trademark of Adobe Systems Inc.

[2] Adobe Systems, *POSTSCRIPT Language Tutorial and Cookbook*, Addison-Wesley, July, 1985.

[3] Adobe Systems, *POSTSCRIPT Language Reference Manual*, Addison-Wesley, July, 1985.

Notational Conventions

This manual uses the following notational conventions:

- **`bold listing font`**

  This font indicates text or code typed at the keyboard.

- `listing font`

  This font indicates information displayed by the computer. It it also used in code examples and textual passages to indicate use of the *C* programming language.

- sans serif font

  This font is used in code examples to indicate use of the POSTSCRIPT language or NeWS extensions.

- **bold font**

  This font is used in textual passages to indicate names of NeWS operators, NeWS types, and system-defined dictionaries.

- *italic font*

  This font is used in code examples and textual passages to indicate user-specified parameters for insertion into programs or command lines. It is also used to indicate special terms or phrases the first time they are used in the text.

1

Introduction

1 Introduction

The X11/NeWS server can be used either by a single computer or by multiple computers linked across a communication network; thus, it is a distributed window system. When the X11/NeWS server is used with multiple computers, an application run by one machine can use the windows displayed by another.

The NeWS interpreted programming language is based on the POSTSCRIPT language. Developed at Adobe Systems, the POSTSCRIPT language is used primarily for specifying the visual appearance of printed documents. A POSTSCRIPT program consists of operations that are sent to a POSTSCRIPT language interpreter residing within a printer; when interpreted, the operations define text, graphics, and page coordinates.

The NeWS language uses POSTSCRIPT language operators to display text and images on a graphics console. Programs are interpreted and executed by the X11/NeWS server, which is resident on the machine to which the graphics console is attached. Importantly, the NeWS language also provides operators and types that are extensions to the POSTSCRIPT language; many of these extensions deal with the interactive and multi-tasking aspects of a window system, which are not handled by the POSTSCRIPT language.

1.1. NeWS Programming: An Overview

This section provides an overview of NeWS programming. Detailed information is provided in later chapters.

The POSTSCRIPT Language

The POSTSCRIPT language is a high level language designed to describe page appearance to a printer. It possesses a wide range of graphics operators. Nevertheless, only about a third of the language is devoted to graphics; the remainder provides a general purpose programming capability.

The POSTSCRIPT language is extensible and thus allows programmers to use the supplied operators to define their own procedures. This extensibility facilitates the creation of modular code, encourages the design of well-structured and comprehensible programs, and helps keep programs small.

NeWS Types

The NeWS language implements all the standard types provided by the POSTSCRIPT language. In addition, the NeWS language provides special types as extensions to the POSTSCRIPT language.

Some of the NeWS type extensions can be accessed as if they were POSTSCRIPT language dictionaries. These objects are known as *magic dictionary* objects.

Magic dictionaries have keys with predefined names. The programmer can change the value associated with many of the keys; other keys are read-only. The programmer can add new keys to magic dictionaries.

Other NeWS type extensions are *opaque* and cannot be accessed as dictionaries. A full description of all NeWS type extensions is provided in Chapter 8, *NeWS Type Extensions*.

NeWS Operators

The NeWS language implements most of the standard operators provided by the POSTSCRIPT language; many of the omitted operators relate to page-description requirements, which are not relevant for a window system. Conversely, the NeWS language provides many operators as extensions to the POSTSCRIPT language; many of these operator extensions relate to interactivity requirements, and many of them exist to create and manipulate the NeWS type extensions.

A full description of all NeWS operator extensions is provided in Chapter 9, *NeWS Operator Extensions*.

The X11/NeWS Server

The X11/NeWS server is not a machine used to serve files; it is a process that can exist on any graphics machine within a network, its function being to interpret and execute programs written in the POSTSCRIPT language and to display the resulting graphics on the screen.

The X11/NeWS server contains multiple *lightweight processes*, some of which communicate with client processes. A lightweight process is not a UNIX® process; it is a process that lives in the server's address space and is scheduled to be run by the server.[1] Each lightweight process can perform operations on the display and can receive messages from the keyboard, the mouse, or another lightweight process. A lightweight process can share data with other lightweight processes. Many lightweight processes can be created with relatively little overhead. Lightweight processes are also known as *NeWS processes*.

Note that the X11/NeWS server is neither a toolkit nor a user interface; it provides neither standards nor defaults for the creation and appearance of windows. The X11/NeWS server simply interprets and executes POSTSCRIPT language operations and NeWS extensions that are specified by the programmer. Different user interfaces can thus be designed entirely by the programmer: written in the POSTSCRIPT language, all can be run on the X11/NeWS server.

Client-Server Communication

The X11/NeWS server communicates with client programs that run either locally or remotely. Clients can send POSTSCRIPT language code to the server. The server runs this code on behalf of the clients.

Typically, a client program contains two main sections. One section, which can be written in C, FORTRAN, or any other language, is used to perform the application's basic computations; this section is executed in the client process. The other section, which must be written in the POSTSCRIPT language, is used to provide corresponding windows or graphics; this section is interpreted by the

[1] UNIX® is a registered trademark of AT&T.

server process. The POSTSCRIPT language section of the client program can be detatched, sent to the server, and executed remotely by means of function calls.

The ability to download POSTSCRIPT programs to the server gives the programmer great freedom in designing the communication protocol and the split in functionality between server and client. The server does not directly notify the client program of events such as mouse manipulation; instead, the server notifies interested lightweight processes, and the client's POSTSCRIPT language code may either handle the information itself or write the information across the connection to the client program. Thus, the way in which the client and server communicate is specified by the POSTSCRIPT language contents of the client application. The POSTSCRIPT language code downloaded by the client program can use any of NeWS' built-in features.

C Client Interface

Most programmers are likely to use C as the language of the client application. Therefore, NeWS provides a special interface facility that supports C client communication. The C client interface, named *CPS*, converts the client's POSTSCRIPT language code into functions callable by the client's C code.

Programmers can also create their own interface facility for use with other languages. The client interface protocol and the C client interface are discussed in Chapter 5, *Client-Server Interface.*

Canvases

A NeWS *canvas* is a region of the screen in which the client application can display information to the user. Canvases provide the basic drawing surfaces in NeWS and are thus the raw material from which windows and menus are created; each window and menu is usually composed of more than one canvas. Canvases need not be rectangular since their boundaries are defined by POSTSCRIPT language paths. When visible on the screen, canvases can overlap. When this occurs, the hidden portion of a canvas can be stored offscreen and redisplayed when the canvas is re-exposed.

A canvas is implemented as a NeWS type extension that can be accessed as a dictionary. Many canvas characteristics can be set by changing the values of the keys in the canvas dictionary. For example, a canvas can be *opaque* or *transparent*, *mapped* or *unmapped*. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. When drawing operations are performed on a mapped canvas, the image is visible on the screen (unless it is overlapped by another canvas); drawing operations can be performed on an unmapped canvas, but the image is not visible on the screen.

Canvases exist in a hierarchy. The background of the screen is the root of the hierarchy and is thus known as the *root canvas.* A canvas can have any number of children; the display of each child canvas is clipped to the edges of its parent. Canvases overlap according to their positions in the hierarchy. When visible on the screen, opaque children obscure their parent. A canvas' children exist in an ordered list that determines their overlapping relationships. For a canvas to be visible on the screen, the canvas and all its ancestors must be mapped.

A canvas can be repositioned in the hierarchy, causing adjustments to the display of any overlapping canvases on the screen. A canvas can also be repositioned horizontally and vertically on the screen, and it can be reshaped and resized.

Each NeWS process can have a *current canvas*, which is the canvas that is manipulated by the drawing operations performed by that process.

The NeWS language provides operator extensions for creating and manipulating canvases. A full account of canvases is provided in Chapter 2, *Canvases*. The canvas dictionary keys are described in Chapter 8, *NeWS Type Extensions*.

The NeWS Imaging Model

The NeWS imaging model, which is essentially that of the POSTSCRIPT language, can be described as a *stencil/paint* model. A *stencil* is an outline specified by an infinitely thin boundary; the boundary can be composed of straight lines, curves, or both. *Paint* is a color, texture, or image that is applied to the drawing surface; the paint appears on the drawing surface within the boundary of the stencil.

Note that the stencil/paint model differs from the *pixel-based* imaging model used by most window systems. The pixel-based model requires that rectangular source and destination areas of pixels be combined using logical operations such as AND, OR, NOT, and XOR. The stencil/paint model allows images of any shape or size, rectangular or non-rectangular, to be specified; it thus provides a more natural and comprehensible way of specifying images.

Events

A NeWS *event* is an object that represents a message between NeWS processes. An event is implemented as a NeWS type extension that can be accessed as a dictionary. Events can transmit any kind of information and thus serve as a general interprocess communication mechanism. Some events report user manipulation of input devices and are therefore known as *input events*.

An event can be generated by the server or by any NeWS process. The server automatically generates input events when the user manipulates the keyboard or mouse. The server also generates events to report when a canvas is damaged, when an object becomes obsolete (see *Memory Management*, below), when a process dies while it is still referenced, and when the mouse pointer leaves one canvas and enters another.

The NeWS language provides operators that allow any NeWS process to create an event and send it into the server's event distribution mechanism. System-generated events are automatically sent into the distribution mechanism as soon as they are generated. After an event enters the distribution mechanism, the server gives a copy of the event to NeWS processes that are interested in the event. The NeWS language provides an operator that allows processes to describe the types of events that interest them; each such description of events that interest a process is known as an *interest*.

A full account of events is provided in Chapter 3, *Events*. The event dictionary keys are described in Chapter 8, *NeWS Type Extensions*.

Memory Management

The X11/NeWS server provides an automatic garbage collection facility that removes objects from virtual memory when the objects are no longer needed. Objects survive as long as they are referenced. If an object's last reference goes away, the server destroys it to reclaim the memory that it occupied.

The NeWS language provides the notion of *soft* references for programs that want to track objects without affecting the lifespan of the objects. A window manager is an example of this type of program. A window manager has references to the canvases that it tracks, but the window manager does not want its references to prevent canvases from being garbage collected. In this type of situation, client programs should use soft references.

If all the references to an object are soft, the object is considered to be *obsolete*. When an object becomes obsolete, the server sends notice, in the form of an event, to all processes that have expressed interest in obsolescence events for that object. The processes should then remove their references to the object so that the server can destroy it.

Note that the server does not count references for all objects. Simple objects such as booleans, numbers, and names are not shared and therefore never have more than one reference. The server only counts references to objects that represent shared resources, such as arrays, dictionaries, canvases, and events.

The NeWS language provides operators that aid in memory management. A full account of the memory management facilities is provided in Chapter 7, *Memory Management.*

Color Support

The NeWS language includes types and operators that provide color support for appropriate displays. A NeWS *color* object consists of either red/green/blue or hue/saturation/brightness components. The NeWS language also provides *colormap* objects, which function as color lookup tables, and *colormapsegment* objects, which are groups of entries within a colormap. Facilities are provided for using *bitmasks* and *planemasks*, which permit colors to be determined according to arithmetic operations.

Full information on all color-related types is provided in Chapter 8, *NeWS Type Extensions.*

Font Support

The server allows bitmap fonts to be defined and placed in the NeWS font library. Cursor fonts and icon fonts can be created, and existing text fonts can be converted into NeWS format. The server provides the commands `convertfont`, `bldfamily`, and `makeiconfont`, which are used in font definition. See the manual pages in the *X11/NeWS Server Guide* for further information.

NeWS font dictionaries are identical to standard POSTSCRIPT language font dictionaries except for one additional key. For a description of the NeWS font dictionary, see Chapter 8, *NeWS Type Extensions.*

1.2. POSTSCRIPT Language Files Used with the Server

In addition to the operator and type extensions that are part of the server itself, the server also provides various POSTSCRIPT language files that support the NeWS programming environment; most of these POSTSCRIPT language files are loaded automatically when the server is initialized. The user can examine the supplied files and modify the procedures that they contain.

This section describes some of the more important POSTSCRIPT language files. Full information on these files is provided in Chapter 10, *Extensibility through POSTSCRIPT Language Files.*

The Window Manager

The POSTSCRIPT language files loaded by the server provide a *default window manager* that allows the user to control the appearance of windows on the screen. The window manager allows the user to move, resize, open, and close windows. Note, however, that the window manager does not actually create the windows; this task is the responsibility of the client application. The default window manager can be replaced if desired. See the *X11/NeWS Server Guide* for more information on the window manager.

Classes

The POSTSCRIPT language files loaded by the server provide support for object-oriented programming; client applications can create objects known as *classes* and *instances*. A class is a template for a set of similar instance objects. A class is essentially a blueprint from which any number of instances can be created. Each instance inherits the characteristics of its class but can override some of these characteristics. Classes and instances are represented as POSTSCRIPT language dictionaries that contain variables and procedures.

NeWS classes belong to a class hierarchy. The root of the hierarchy is class **Object**; class **Object** is implemented by the server, and the other classes in the hierarchy are provided by the client or by a toolkit.

Any class in this system can have *subclasses*, each of which inherits the characteristics of its *superclass*. A subclass can add new characteristics and can override its inherited characteristics. A subclass can also inherit characteristics from more than one branch of the class tree, a feature known as *multiple inheritance*.

The class system is especially useful for defining user interfaces. For example, class Canvas might be a subclass of class **Object**, and class Canvas might have subclasses such as Menu, Scrollbar, Frame, and Window.

Information on the class system is provided in Chapter 4, *Classes*.

Debugging

The server provides a *debugging* facility that allows the user to set breakpoints and print to debugging output windows. The POSTSCRIPT language file containing the debugger code is not loaded when the server is initialized; a command must be given to load this file.

Full information on using the debugger is provided in Chapter 6, *Debugging*.

2

Canvases

2

Canvases

A NeWS *canvas* is a region of the screen in which the client application can display information to the user. Canvases provide the basic drawing surfaces in NeWS and are thus the raw material from which windows and menus are created; each window and menu is usually composed of more than one canvas. Canvases need not be rectangular since their boundaries are defined by POSTSCRIPT language paths. When visible on the screen, canvases can overlap. When this occurs, the hidden portion of a canvas can be stored offscreen and redisplayed when the canvas is re-exposed.

A canvas is implemented as a NeWS type extension that can be accessed as a dictionary. Many canvas characteristics can be set by changing the values of the keys in the canvas dictionary. For example, a canvas can be *opaque* or *transparent*, *mapped* or *unmapped*. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. When drawing operations are performed on a mapped canvas, the image is visible on the screen (unless it is overlapped by another canvas); drawing operations can be performed on an unmapped canvas, but the image is not visible on the screen.

Canvases exist in a hierarchy. The background of the screen is the root of the hierarchy and is thus known as the *root canvas*. A canvas can have any number of children; the display of each child canvas is clipped to the edges of its parent. Canvases overlap according to their positions in the hierarchy. When visible on the screen, opaque children obscure their parent. A canvas' children exist in an ordered list that determines their overlapping relationships. For a canvas to be visible on the screen, the canvas and all its ancestors must be mapped.

A canvas can be repositioned in the hierarchy, causing adjustments to the disp of any overlapping canvases on the screen. A canvas can also be moved on the screen, and it can be reshaped and resized.

Each NeWS process can have a *current canvas*, which is the canvas that is manipulated by the drawing operations performed by that process.

This chapter describes canvases and shows how they can be used.

The canvastype Extension

Each canvas is an object of type **canvastype**, which is a NeWS extension to the POSTSCRIPT language. Each **canvastype** object can be accessed as a POSTSCRIPT language dictionary. The values of the dictionary keys determine the properties of the canvas. A canvas dictionary includes keys for specifying the following:

- Ancestor and sibling relationships between canvases (**TopCanvas, Bottom-Canvas, CanvasAbove, CanvasBelow, TopChild, Parent**)
- The appearance of canvases on the screen (**Transparent, Mapped**)
- The handling of canvas storage (**Retained, SaveBehind**)
- The event management properties of the canvas (**EventsConsumed, Interests**)
- The color properties of the canvas (**Color, Colormap, Visual, VisualList**)
- The cursor associated with the canvas (**Cursor**)
- Properties for keeping a canvas in shared memory (**SharedFile, RowBytes**)
- X11-related properties (**OverrideRedirect, BorderWidth, UserProps, XID**)
- The grabbed state of a canvas (**Grabbed, GrabToken**)

The keys are discussed in detail throughout this chapter; a full syntactic description of each key is also provided in Chapter 8, *NeWS Type Extensions*.

Canvas Operators

NeWS includes a variety of operator extensions to be used on canvases. The operators provide the following functionality:

- Creating canvas objects and overlays (**buildimage, createdevice, createoverlay, newcanvas**)
- Changing ancestor and sibling relationships between canvases (**canvastobottom, canvastotop, insertcanvasabove, insertcanvasbelow**)
- Defining canvas shapes and paths (**clipcanvas, clipcanvaspath, eoclipcanvas, eoreshapecanvas, reshapecanvas**)
- Reading and writing canvases to files (**eowritecanvas, eowritescreen, imagecanvas, imagemaskcanvas, readcanvas, writecanvas, writescreen**)
- Determining and specifying canvas locations (**getcanvaslocation, movecanvas**)
- Specifying the current canvas (**setcanvas**)

The operators are described throughout this chapter; a list of the operators is provided for quick reference in Appendix A, *NeWS Operators*. A syntactic analysis and description of all NeWS operators is provided in Chapter 9, *NeWS Operator Extensions*.

2.1. Coordinate System Overview

In the standard use of the POSTSCRIPT language, a user coordinate system is associated with the page and a device coordinate system is associated with the printer. A *current transformation matrix*, or CTM, contains the current transformation from user coordinates to device coordinates. The CTM can be changed at any time with operators such as **scale**, **rotate**, or **translate**.

In NeWS, each canvas represents a separate "user space" with its own coordinate system, and the device space corresponds to the screen rather than to a printer. A current transformation matrix is still used to store the current transformation between the user and device coordinate systems, but in NeWS, each process has its own CTM as a part of its graphics state. A process' current coordinate system is specified by its CTM. When a new process is created, it inherits its CTM (along with the rest of its graphics state) from its parent.

Each NeWS canvas has a *default coordinate system* determined by its *default transformation matrix*. A canvas' default transformation matrix specifies the initial transformation from the canvas' coordinate system to the screen's coordinate system. After a new, empty canvas is created with **newcanvas**, the canvas' shape and default coordinate system should be set with **reshapecanvas**. The **reshapecanvas** operator sets the canvas' shape to be the same as the current path and sets the canvas' default coordinate system to be the same as the CTM.

When a canvas is made the current canvas with the **setcanvas** operator, the CTM is set to that canvas' default transformation matrix. The CTM can then be changed with standard POSTSCRIPT language operators. To change an existing canvas' default transformation matrix and shape, simply set the CTM and current path to the desired values and execute **reshapecanvas**.

These coordinate system definitions are illustrated in this chapter's examples.

2.2. Creating and Displaying Canvases

This section discusses how canvases can be created, shaped, and mapped to the display. It also provides an introduction to using the **canvastype** type extension.

Creating Canvases

NeWS canvases exist in a hierarchy; thus, each canvas has a parent and can have one or more children.

When NeWS is initialized, the **createdevice** operator is called to create the background of the screen (note that the **createdevice** operator should not normally be used by the programmer). This background is known as the *root canvas* or *framebuffer*; it can be accessed by means of a global variable named **framebuffer**. Any canvas that you wish to create immediately on top of this background must have **framebuffer** specified as its parent.

NeWS provides the following operator for creating a canvas with a specified parent:

pcanvas newcanvas ncanvas

This operator creates a new canvas whose parent is *pcanvas*.

If **framebuffer** is used as the *pcanvas* argument, the new canvas is *opaque* by default. If the parent is specified as some other canvas, the new canvas is *transparent*. Detailed information on canvas transparency and other factors affecting canvas appearance is provided later in this chapter.

The following example uses **newcanvas** to create a new canvas that has the framebuffer specified as its parent.

```
/FirstCanvas framebuffer newcanvas def
```

FirstCanvas is opaque by default, since the framebuffer is its parent.

Shaping Canvases

All coordinates in NeWS are measured with reference to the *origin* of a specified canvas, which is often the lower-left corner of the canvas. Methods exist both for specifying the origin of a canvas and for specifying the canvas from whose origin coordinates are measured; these are discussed throughout this chapter.

NeWS allows you to shape canvases according to the current path. The following operator reshapes a canvas.

canvas reshapecanvas –

This operator sets the shape of *canvas* to be the same as the current path. It also sets the canvas' default transformation matrix to be the same as the current coordinate system. If the canvas that is being reshaped is the current canvas, this operator sets the current clipping path (in the graphics state) to be the same as the canvas' new shape.

The following example uses **reshapecanvas** to establish a shape and default coordinate system for the canvas defined in the previous example:

```
newpath                              % Define a path to which the
0 0 moveto                           % new canvas can be shaped.
0 250 lineto
250 250 lineto
250 0 lineto
closepath

FirstCanvas reshapecanvas            % Reshape the canvas to the
                                     % current path.
```

NOTE *NeWS also provides an operator named* **eoreshapecanvas**. *This operator is identical to* **reshapecanvas** *except that it uses the even-odd winding rule, rather than the non-zero winding rule, to interpret the specified path. For information on winding rules, see the POSTSCRIPT Language Reference Manual. For an analysis of* **eoreshapecanvas**, *see Chapter* 9, NeWS Operator Extensions.

Setting the Current Canvas

NeWS supports the concept of a *current canvas*. Each NeWS process can have a current canvas as part of its graphics state. Many NeWS canvas operations do not take a *canvas* argument, but simply use the current canvas. To set the current canvas, use the following operator:

canvas **setcanvas** –

The operator sets *canvas* to be the current canvas. It also sets the current coordinate system to be the same as the canvas' default coordinate system. The current coordinate system can then be changed by calls to **scale, rotate,** and **translate.** The **setcanvas** operator sets the current clipping path to be the same as the canvas' shape.

The following example demonstrates how to set the current canvas:

```
FirstCanvas setcanvas
```

Mapping Canvases to the Screen

NeWS does not provide an operator for mapping canvases; instead, it allows you to map canvases by setting the **Mapped** key of the **canvastype** dictionary to **true.** This causes the canvas to be visible on the screen within the borders of its parent, provided that the following conditions are fulfilled:

- All of the canvas' ancestors are also mapped.
- The canvas is not clipped away by its parent or any overlapping canvases.

Even when the canvas is mapped, it might not be noticed on the screen if no drawing operations have been performed on it.

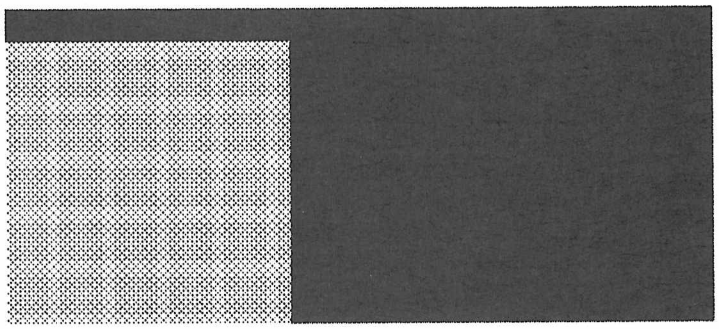
To fill a canvas with a color, use the **fillcanvas** operator; this operator is included in the POSTSCRIPT language extensibility files that NeWS provides. The operator takes a single argument, which can be an integer or a color; see Chapter 10, *Extensibility through POSTSCRIPT Language Files* for a syntactic analysis.

To retrieve and establish values for any read-write NeWS dictionary key, use the POSTSCRIPT language operators **get** and **put** respectively. This is demonstrated by the following example:

```
FirstCanvas /Mapped get ==
false

FirstCanvas /Mapped true put            % Make the canvas visible and
FirstCanvas setcanvas                   % paint it.
.9 fillcanvas
```

When FirstCanvas has been mapped to the screen, it appears in this example at the bottom-left corner of the framebuffer. This is illustrated in the following figure:

Figure 2-1 *Canvas Mapped at 0,0*



NOTE *NeWS does not provide an operator for destroying a canvas; even when unmapped, a canvas continues to exist. A canvas is destroyed only when the last reference to the canvas is removed (see Chapter 7, Memory Management).*

2.3. Manipulating Canvases

This section discusses how canvases can be manipulated on the screen. It describes how to move canvases and discusses the concepts of *transparency*, *retaining*, and *damage*.

Moving Canvases

The display of a canvas is clipped to its parent's boundaries; thus, if a canvas is moved or reshaped so that parts of the canvas fall outside of its parent's boundaries, those parts of the canvas do not appear on the screen when the canvas is mapped.

NeWS provides the following operator for moving canvases:

x y **movecanvas** –
x y canvas **movecanvas** –
If no *canvas* argument is specified, this operator moves the current canvas so that the origin of its default coordinate system is at the coordinates *x* and *y*, where (*x*, *y*) is a vector from the origin of the parent canvas' default coordinate system to the origin of the repositioned current canvas' coordinate system, measured in units of the current coordinate system.

If the *canvas* argument is specified, the operator moves that canvas so that the origin of its default coordinate system is at the coordinates *x* and *y* in the current coordinate system.

NOTE *If a canvas obscures an unretained canvas and is then moved, damage occurs on the unretained canvas. Detailed information on damage and retained canvases is provided in the following section.*

The following operator returns the coordinates of a canvas:

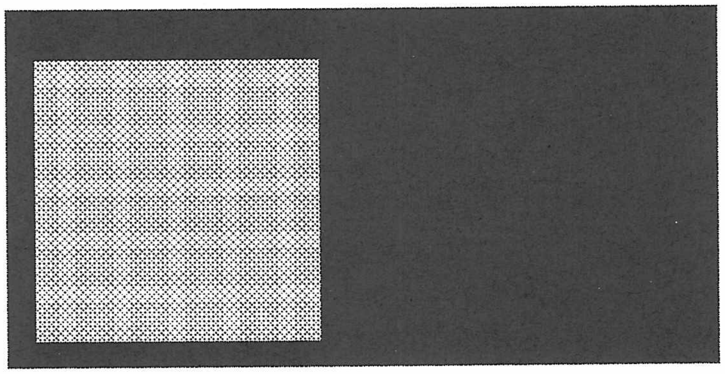
canvas **getcanvaslocation** x y
The operator returns two integers, which specify the *x* and *y* location of the origin of *canvas*' default coordinate system. This location is specified relative to the origin of the current coordinate system (rather than of the parent canvas); thus, the coordinates may be returned as either negative or positive integers.

In the following example, a canvas is moved and its coordinates are returned:

```
25 25 movecanvas                    % Move the current canvas relative
                                    % to its parent.

framebuffer setcanvas
FirstCanvas getcanvaslocation       % Return the location of FirstCanvas.
pstack
25 25
clear
```

The appearance of FirstCanvas is now as follows:

Figure 2-2 *Canvas Mapped at 25,25*



Transparent and Opaque Canvases

An *opaque* canvas visually hides all canvases underneath it; a transparent canvas does not. When you create a canvas whose parent is any canvas other than the framebuffer, the new canvas is transparent by default. If drawing operations are performed on a transparent canvas, the drawn images appear on the canvas(es) beneath the transparent canvas (its parent or any siblings that are beneath it); thus, if the transparent canvas is unmapped, the drawn images remain. A transparent canvas may define screen areas that are sensitive to input.

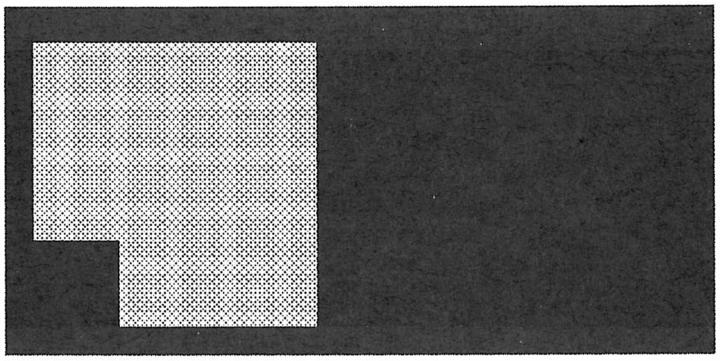
To create an opaque canvas whose parent is not the framebuffer, you must set the new canvas' **Transparent** key to **false**. The following `psh` example creates a canvas whose parent is the canvas FirstCanvas , which was created in the previous example.

Note that this example uses the **rectpath** utility, which is provided by the POSTSCRIPT language extensibility files. The **rectpath** utility takes four numbers as arguments: the *x* and *y* location of the rectangle origin, the *width* of the rectangle, and the *height* of the rectangle. The **rectpath** utility adds the rectangle to the current path. For a complete definition of **rectpath**, see Chapter 10, *Extensibility through POSTSCRIPT Language Files.*

```
FirstCanvas setcanvas
/SecondCanvas FirstCanvas
newcanvas def                        % Define a canvas.

newpath                              % Define a path to which the
0 0 75 75 rectpath                   % new canvas can be shaped.
SecondCanvas reshapecanvas           % Reshape the new canvas.
SecondCanvas /Transparent get        % Prove the canvas is transparent.
pstack
true

clear

SecondCanvas /Transparent false put  % Make the canvas opaque.
SecondCanvas /Mapped true put        % Map the canvas.
SecondCanvas setcanvas               % Paint the canvas.
0 fillcanvas
```

The appearance of SecondCanvas is as follows:

Figure 2-3 *A Mapped Child Canvas*

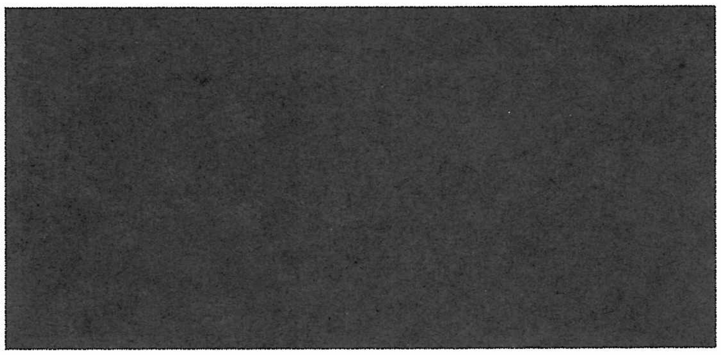


When a parent canvas is made transparent, its opaque children are not affected and remain opaque. This is demonstrated by the following example:

```
FirstCanvas /Transparent true put    % Make the parent transparent; the
                                     % opaque child remains opaque.
```

This example is illustrated in the following figure:

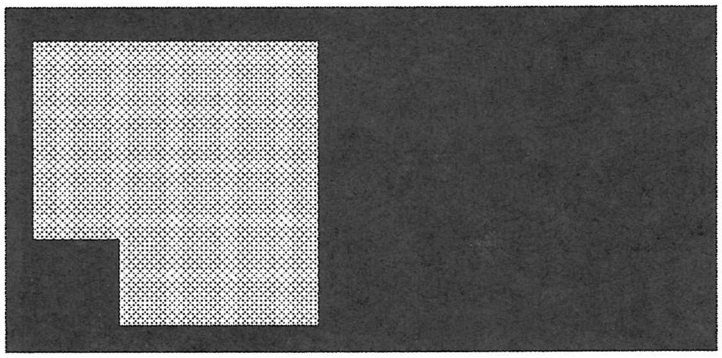
Figure 2-4 *Parent Canvas Made Transparent*



The following code makes the parent opaque again and paints it, thus restoring FirstCanvas and SecondCanvas to their previous appearance:

```
FirstCanvas /Transparent false put
FirstCanvas setcanvas .9 fillcanvas
```

Figure 2-5 *Parent Canvas Made Opaque and Repainted*



Retained and Non-Retained Canvases

The **canvastype** dictionary has a key named **Retained** that can be set to **true** or **false** and specifies whether or not a canvas is *retained.* If a canvas is retained, any portion of its visible surface that becomes obscured by another canvas is automatically saved offscreen. The saved portion is restored to the screen automatically when no longer obscured; thus, the canvas does not receive damage and does not need to be redrawn.

A transparent canvas does not have its own retained image; instead, it shares the retained image of its parent.

If a canvas is unretained, *damage* occurs when another canvas, by which it was previously obscured, is moved or unmapped; the damage takes the form of an after-image of the moved canvas. Damage can also occur in other situations, such when an unretained canvas is mapped to the screen. Note that a transparent canvas never receives damage; instead, damage may be received by the canvas beneath the transparent canvas.

A client must be prepared for damage even on a retained canvas: a retained canvas can and will receive damage, although less frequently than will an unretained canvas. For example, a retained canvas may receive damage when it is reshaped.

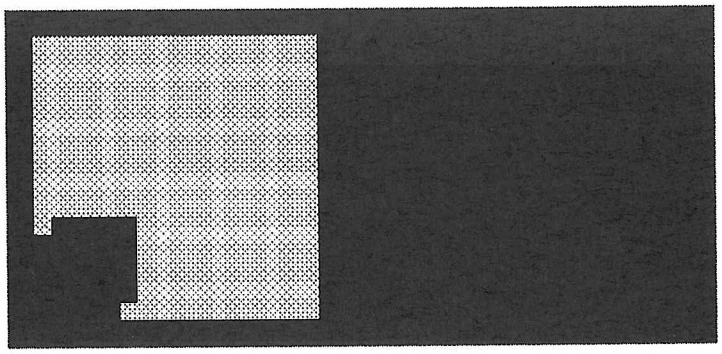
NOTE *Each system has a retain threshold that specifies the number of bits per pixel below which a canvas has its* **Retained** *key automatically set to* **true**. *However, if your application desires that a canvas be retained, you should always set the* **Retained** *key explicitly.*

The following example demonstrates the effects of setting the **Retained** key:

```
FirstCanvas /Retained false put       % Make a parent canvas unretained.
SecondCanvas setcanvas
15 15 movecanvas                      % Move the child canvas over the
                                      % parent; damage occurs to the
                                      % parent.
```

The damage caused by moving SecondCanvas is illustrated as follows:

Figure 2-6 *Damage on Unretained Canvas*



If the damaged parent canvas is repainted and then retained, the child canvas can be moved over its surface without damage occurring:

```
FirstCanvas setcanvas                 % Paint the parent, the appearance
.9 fillcanvas                         % of the child is not affected.

FirstCanvas /Retained true put        % Retain and paint the parent.
.9 fillcanvas

SecondCanvas setcanvas                % Move the child over the retained
25 25 movecanvas                      % parent; no damage occurs.
```

NOTE *Retaining canvases can be extremely costly in terms of memory, particularly on color displays.*

Further Information on Damage

The NeWS server considers a canvas to be damaged if all or part of its image is incorrect and needs to be redrawn. Damage can occur in the following ways:

- An unretained canvas is damaged when a canvas by which it was previously obscured is moved away.
- An unretained canvas is damaged when first mapped to the screen.
- An unretained canvas is damaged when its **Retained** key is set to **true** (thus changing it to retained) while part of the canvas is not visible.
- A retained or unretained canvas is damaged when is it reshaped.

When a canvas is initially damaged, the NeWS server automatically sends a damage event to processes interested in damage on that canvas; a damage event has **/Damaged** in its **Name** field and a copy of the affected canvas in its **Canvas** field. After receiving a damage event, the client program should repair the damage by redrawing the damaged parts of the canvas. If the client does not immediately repair the canvas and damage continues to occur, the NeWS server sends no additional damage events to the client. Instead, the server maintains and updates a record of all the damage that has occurred to the client. Eventually, the client should request a copy of this record and repair all damage (for information on doing this, see the description of the **damagepath** operator in Chapter 9, *NeWS Operator Extensions*).

The SaveBehind Key

The **SaveBehind** key of the **canvastype** dictionary can be used to prevent damage from occurring to other canvases. When the key is set to **true**, NeWS saves the values of the pixels that the canvas obscures when it is mapped. Even if the pixels belong to unretained canvases, they can be restored directly to the screen when the canvas that obscures them is removed.

The **SaveBehind** key is useful for pop-up menus and other canvases that are small and are not required to be visible for long; when used with such canvases, the key can greatly enhance server performance.

The **SaveBehind** key is demonstrated by the following example:

```
SecondCanvas /Mapped false put          % Unmap the child.

FirstCanvas /Retained false put         % Make the parent unretained.

SecondCanvas /SaveBehind true put       % Specify use of SaveBehind.

SecondCanvas /Mapped true put           % Remap the child.

SecondCanvas setcanvas                  % Make the child current, paint
0 fillcanvas                            % it, and unmap it; no damage
SecondCanvas /Mapped false put          % occurs to the parent.
```

2.4. Parent, Child, and Sibling Canvases

This section discusses the parent/child and sibling relationships that exist between canvases.

The Sibling List

When a parent canvas has multiple children, the children are automatically arranged in a *sibling list*. The list controls the appearance of the siblings when they are visible on the screen and are made to overlap. By default, the most recently created child becomes the *top sibling* in the list; thus, if all siblings are visible on the screen and overlap, the top sibling covers all others; the bottom sibling is covered by all others. When the **newcanvas** operator is called, the created canvas becomes the top child of its parent.

You can change the list-position of a sibling by specifying values for the following keys of the **canvastype** dictionary:

- **CanvasAbove**

  This key specifies the canvas that is immediately above this canvas in the sibling list; if no such canvas exists, the value of the key is **null**. The value of this key can be set to any sibling; thus, this canvas is inserted in the list at a position directly below the specified sibling.

- **CanvasBelow**

  This key specifies the canvas that is immediately below this canvas in the sibling list; if no such canvas exists, the value of the key is **null**. The value of this key can be set to any sibling; thus, this canvas is inserted in the list at a position directly above the specified sibling.

NeWS also provides the following operators, which allow you to manipulate the sibling list:

canvas **canvastobottom** –
Moves the *canvas* to the bottom of its list of siblings.

canvas **canvastotop** –
Moves the *canvas* to the top of its list of siblings.

canvas x y **insertcanvasabove** –
Inserts the current canvas into the list at the position immediately above *canvas*.

canvas x y **insertcanvasbelow** –
Inserts the current canvas into the list at the position immediately below *canvas*.

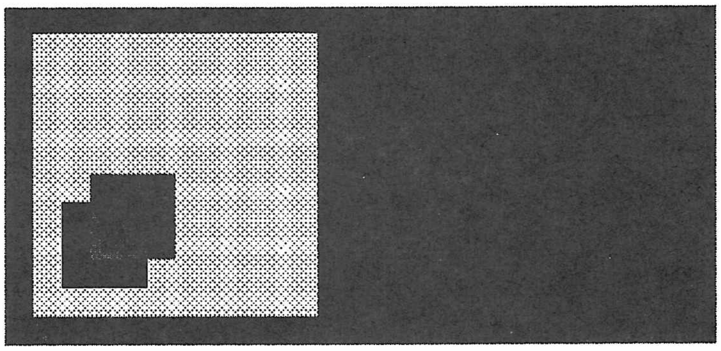
For the operators **insertcanvasabove** and **insertcanvasbelow**, the coordinates *x* and *y* are computed with reference to the origin of the default coordinate system of the parent of the current canvas (as described for the form of the **movecanvas** operator that takes no canvas argument). The current canvas must be a sibling of *canvas*.

The following example creates ThirdCanvas , which is a sibling to the canvas SecondCanvas (defined in a previous example); by default, the new canvas is placed at the top of the sibling list and thus obscures its sibling. The **insertcanvasabove** operator is then used to reverse the position of the canvases in the list; thus, SecondCanvas obscures the new canvas.

```
FirstCanvas /Retained true put
SecondCanvas /Mapped true put
0 fillcanvas
SecondCanvas /Retained true put
FirstCanvas setcanvas
/ThirdCanvas FirstCanvas newcanvas def % Define and display a sibling
                                       % of SecondCanvas.
newpath 0 0 75 75 rectpath
ThirdCanvas reshapecanvas
ThirdCanvas /Transparent false put
ThirdCanvas /Retained true put
ThirdCanvas setcanvas
.4 fillcanvas
50 50 movecanvas
ThirdCanvas /Mapped true put           % ThirdCanvas obscures
                                       % SecondCanvas.
```

The appearance of ThirdCanvas is illustrated as follows:

Figure 2-7 *Younger Sibling Obscuring Elder*

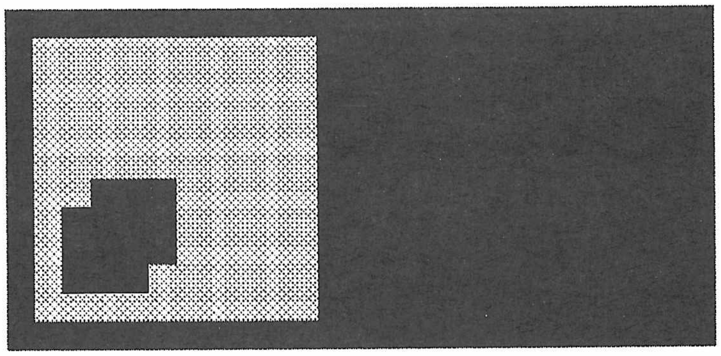


The following code inserts SecondCanvas above ThirdCanvas :

```
SecondCanvas setcanvas                 % Insert SecondCanvas above
ThirdCanvas 25 25 insertcanvasabove    % ThirdCanvas.
```

The appearance of the canvases is now as follows:

Figure 2-8 *Elder Sibling Made to Obscure Younger*



The **insertcanvasbelow** operator can similarly be used to change sibling positions in the list:

```
SecondCanvas setcanvas                    % Insert SecondCanvas below
ThirdCanvas 25 25 insertcanvasbelow       % ThirdCanvas.
```

Establishing a New Parent

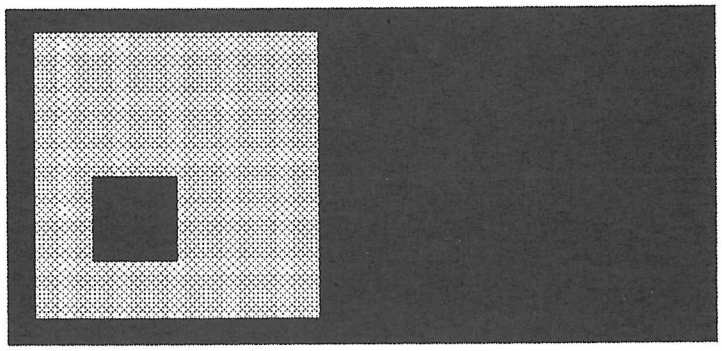
NeWS allows you to specify a new parent for a canvas by setting the value of the **Parent** key in the **canvastype** dictionary.

This is shown by the following example:

```
SecondCanvas /Parent ThirdCanvas put      % Make ThirdCanvas the parent of
20 20 movecanvas                          % SecondCanvas
```

The canvases now appear as follows (note that SecondCanvas is now clipped where it exceeds the boundaries of ThirdCanvas):

Figure 2-9 *Modified Parenthood Between Canvases*



2.5. Overlay Canvases

NeWS allows you to create *overlay canvases*. An overlay canvas, which can only be created over an existing non-overlay canvas, is always transparent. However, when graphic objects are drawn on an overlay, they appear on the overlay itself, rather than on the canvas below.

Overlays are intended for use in transient or animated drawing procedures, such as the creation of "rubber-band" boxes, which expand or contract according to mouse movement when a user is resizing a window.

The following operator is provided for creating overlays:

canvas **createoverlay** canvas

The *canvas* argument must be an existing canvas; the canvas object returned is the created overlay. Note that the overlay is not a child of the specified *canvas*; it is considered a part of that canvas.

Other features of overlays are as follows:

- Each non-overlay canvas (whether transparent or opaque) can possess one overlay canvas only.
- An overlay canvas cannot receive any events. If you express interest on an overlay, the interest is placed on the prechild interest list of the canvas over which the overlay was created.
- An overlay never receives damage; therefore, it never requires repainting.
- An overlay cannot have a parent, nor can it have children.
- If an overlay's corresponding non-overlay canvas has children, these may have their own overlays. A canvas' overlay appears above the overlays of the canvas' children.
- If a canvas possesses an overlay, any subsequent attempt to create an overlay of the canvas returns the existing overlay.
- An overlay cannot be reshaped; attempting to reshape an overlay produces no result. An overlay always has the shape of its associated non-overlay canvas.
- An overlay cannot be possessed by more than one non-overlay, nor can it change owners.
- An overlay should not be specified as mapped or unmapped; it should appear in accordance with the state of its associated non-overlay. Programmers should not attempt to change the keys in the overlay's dictionary.

Drawing on Overlays

Due to the way in which overlays are implemented on some machines, performance problems may occur if too many objects are drawn on an overlay.

The current color is usually ignored when drawing operations are performed on overlays; this is a deliberate feature to allow the implementation of overlays using various procedures on different kinds of hardware.

2.6. Canvas Clipping Operators

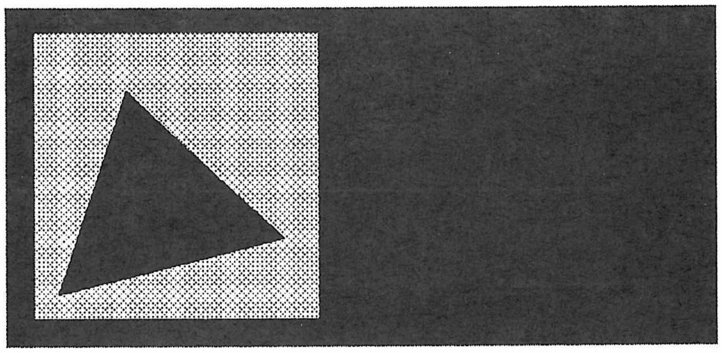
NeWS provides operators that perform clipping operations on canvases. These operators are similar to the clipping operators described in the *POSTSCRIPT Language Reference Manual*, except that they specify paths that NeWS considers only in relation to the current canvas. These operators, **clipcanvas, clipcanvaspath,** and **eoclipcanvas**, are described in Chapter 9, *NeWS Operator Extensions*. The clipping operators are typically used to limit the portion of the canvas painted during damage repair.

The following example demonstrates the **clipcanvas** operator:

```
SecondCanvas /Mapped false put
ThirdCanvas /Mapped false put

FirstCanvas setcanvas

newpath                         % Define a path.
  20 20 moveto
  220 70 lineto
  80 200 lineto
closepath

clipcanvas
0 fillcanvas                    % The fillcanvas operation is performed—
                                % not on the entire canvas, which is the
                                % default, but on the area within the
                                % specified path.
```

The appearance of FirstCanvas is now as follows:

Figure 2-10 *Results of Canvas Clipping Operation*



2.7. Cursors

The **canvastype** dictionary contains a **Cursor** key, which specifies the cursor object that is used whenever the mouse is positioned over the canvas. When a canvas is created with **newcanvas**, it inherits the **Cursor** value of its parent.

A cursor is composed of a *cursor image* and a *mask image*; the complete cursor is produced by superimposing these two images. The mask and cursor images each have three attributes: a font, a character in the font, and a color. The default color for the cursor image is black, while the default color for the mask image is white. The two images are superimposed by aligning the origins associated with their characters.

Each cursor has a *hot spot*, which is the pixel coordinate to which the mouse points. The hot spot resides at the superimposed origin of the mask and cursor images.

See Chapter 8, *NeWS Type Extensions* for further information on cursors and the values to which the **Cursor** key can be set.

2.8. Canvases, Files, and Imaging Procedures

NeWS provides operators that allow you to save canvases in files and read them back into NeWS; it also provides operators that image canvas objects to the display. This section describes these operators and how they can be used.

Writing Canvases to Files

NeWS provides the following operator, which allows you to write a canvas to a file:

file *or* string **writecanvas** –
This operator writes the current canvas either to a file object (specified by *file*) or to a file in the server's file name space (specified by *string*). The operator creates a rasterfile that contains an image of the region outlined by the current path in the current canvas. If the current path is empty, the whole canvas is written.

The **writecanvas** operator uses the non-zero winding rule; see the *POSTSCRIPT Language Reference Manual* for information. To write a canvas to a file using the even-odd winding rule, use **eowritecanvas**.

The **writecanvas** operator is demonstrated by the following example:

```
FirstCanvas setcanvas
(canvasfile) writecanvas                % Write the canvas to a file.
```

Reading Canvases from Files

The following operator reads a canvas from a file:

string *or* file **readcanvas** canvas
The operator reads a raster file into a newly created *canvas*. The raster file can be specified either as a file or as a string that is the name of a file in the server's file name space. The created canvas is retained and opaque; it has the depth specified in the raster file, has no parent, and is not mapped. **readcanvas** sets the default coordinate system of the canvas so that the canvas' four corners correspond to the unit square.

If the filename specified by the string cannot be found, an `undefined-filename` error is generated. If the file cannot be interpreted as a raster file, an `invalidaccess` error is generated.

Note that a canvas read into NeWS with this operator cannot be mapped to the display; any attempt to do this results in an **invalidaccess** error. The canvas must be used as source for the **imagecanvas** operator, which is described as follows:

canvas imagecanvas –

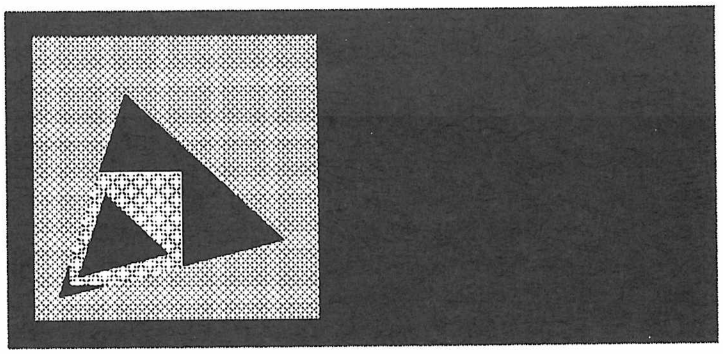
The operator paints the *canvas* argument onto the current canvas. When *canvas* is rendered, the unit square is transformed to the same orientation and scale as the unit square in the current transformation matrix. (Note that this operator is similar to the **image** operator provided by the POSTSCRIPT language.)

The **readcanvas** and **imagecanvas** operators are demonstrated by the following example, in which the canvas saved in the previous example is imaged to the screen:

```
30 30 translate
100 100 scale
/FileCanvas (canvasfile) readcanvas def
FileCanvas imagecanvas
```

The appearance of FirstCanvas is now as follows:

Figure 2-11 *Imaged Canvas*



Other File-Related Operators

NeWS also provides the operators **writescreen** and **eowritescreen**, each of which creates a raster file that contains a snapshot of the screen, clipped to the current path in the current canvas. The operators **writecanvas** and **eowritecanvas** each create a raster file that contains an image of the region outlined by the current path in the current canvas.

See Chapter 9, *NeWS Operator Extensions* for a complete analysis of these operators.

Imaging

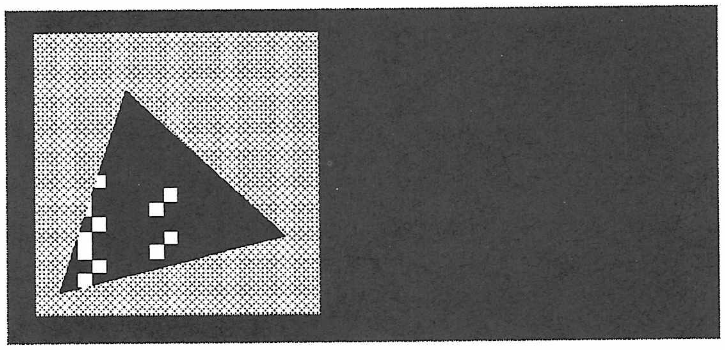
The NeWS operator **buildimage** provides functionality similar to that of the **image** operator provided by the POSTSCRIPT language, using the binary representation of a specified string to create a sampled image as a canvas object. The canvas object can then be imaged to the screen with the **imagecanvas** operator.

The **buildimage** operator is described fully in Chapter 9, *NeWS Operator Extensions*. The following example demonstrates how it can be used:

```
FirstCanvas setcanvas
0 fillcanvas
/Design 8 8 1 [8 0 0 8 0 0] (A B) buildimage def
25 25 translate                    % Specify appropriate
100 100 scale                      % coordinates and scale.

Design imagecanvas                 % The image appears within
                                   % the clipping area of FirstCanvas.
```

The appearance of FirstCanvas is now as follows:

Figure 2-12 *Canvas Imaged with buildimage Operator*



2.9. Other Dictionary Keys

Some of the **canvastype** dictionary keys pertain to NeWS features that are described fully elsewhere in this guide. This section summarizes these keys and their functionality.

Events

The keys **EventsConsumed** and **Interests** control the behavior of the canvas with reference to *events*. These keys are described in Chapter 3, *Events* and Chapter 8, *NeWS Type Extensions*.

Color

The keys **Color**, **Colormap**, **Visual**, and **VisualList** manage the color requirements of canvases that are displayed on the appropriate hardware. These keys are described in Chapter 8, *NeWS Type Extensions*.

X-Specific Features

The keys **OverrideRedirect**, **BorderWidth**, **UserProps**, and **XID** are used only for canvases created by X11. These keys are described in Chapter 8, *NeWS Type Extensions*.

Grab State

The keys **Grabbed** and **GrabToken** are used to set and inspect the grabbed state of a canvas. These keys are described in Chapter 8, *NeWS Type Extensions*.

File Sharing

The keys **SharedFile** and **RowBytes** are used to map canvases to files; the keys are described in Chapter 8, *NeWS Type Extensions*. Note that the ability to map a canvas to a file is operating system dependent and may not be present in the server.

3

Events

3

Events

An *event* is an object that represents a message between NeWS processes. An event can be generated by the server or by any NeWS process, and an event can be delivered to any NeWS process. Events that originate from the server are known as *system-generated* events; events that originate from NeWS processes are known as *process-generated* events. Events can transmit any kind of information and thus serve as a general interprocess communication mechanism. Some system-generated events report user manipulation of input devices and are therefore known as *input events*. An event is implemented as a NeWS type extension that can be accessed as a dictionary.

NeWS provides an operator, **createevent**, that allows a process to create an event object. The newly created event dictionary contains keys with system-supplied names and initial values of null or zero. The process can then give the desired values to the keys and send the event into distribution. A process sends an event into the server's distribution mechanism with the **sendevent** operator. System-generated events are automatically sent into distribution as soon as they are generated. The server's distribution mechanism accumulates events in a *global event queue* and distributes a copy of each event to NeWS processes that are interested in receiving the event.

A process indicates its interest in receiving a certain type of event by constructing that type of event and passing it as an argument to the **expressinterest** operator. An event object used in this way is known as an *interest*. A process' interests serve as templates that tell the server what types of events the process wants to receive.

This chapter provides a full account of events.

The eventtype extension

Each NeWS event is of type **eventtype** and can be accessed as a POSTSCRIPT language dictionary. An event dictionary contains keys that describe the following:

- The identity and matching of events (**Action**, **Name**, **Serial**, **Process**)
- The location or destination of events (**Canvas**, **Coordinates**, **XLocation**, **YLocation**)
- The time at which an event is to be distributed (**TimeStamp**)

- Whether an event is in the server's global event queue (**IsQueued**)
- The interest that matched an event (**Interest**)
- The characteristics of interests (**Exclusivity, IsInterest, IsPreChild, Priority**)
- Keyboard status with reference to events (**KeyState**)
- Additional miscellaneous information (**ClientData**)

The keys are discussed in detail throughout this chapter; a full syntactic description of each key is also provided in Chapter 8, *NeWS Type Extensions*.

Event Operators

NeWS includes a variety of operator extensions to be used on events. The operators provide the following functionality:

- Creating events (**createevent**)
- Sending events into distribution (**sendevent**)
- Enabling and disabling reception of events (**awaitevent, expressinterest, revokeinterest**)
- Manipulating the event distribution mechanism (**blockinputqueue, recallevent, redistributeevent, unblockinputqueue**)
- Performing miscellaneous operations (**countinputqueue, geteventlogger, lasteventkeystate, lasteventtime, lasteventx, lasteventy, postcrossings, seteventlogger**)

3.1. Overview of Event Distribution

The distribution of an event consists of four main steps:

1. Generation of the event.

   An event is created by the server or by any NeWS process. A process creates an event with the **createevent** operator.

2. Delivery of the event to the server's global event queue.

   A process sends an event to the global event queue with the **sendevent** operator. System-generated events are automatically sent to the global event queue after the server creates them.

   The events in the global queue are sorted according to the value of their **TimeStamp** keys. When the server generates an event, the current time is stored in the event's **TimeStamp** key. Other events have whatever **TimeStamp** value is specified by the process that creates them. An event is never delivered before the time indicated in its **TimeStamp** key. Therefore, processes can specify that an event be delivered at some time in the future.

3. Distribution of the event to interested processes.

   Delivery of an event is initiated whenever the event at the head of the global event queue has a **TimeStamp** that is less than or equal to the server's current time. When this occurs, the event is removed from the queue and is compared with the interests to locate matches. An event is not necessarily compared to all the interests; the value of the event's **Canvas** key determines which interests are compared to the event. (The search procedure is described in detail later in this chapter.)

When an event is compared to an interest, the server attempts to match four of the dictionary keys in the event to the same four keys in the interest: the **Name, Action, Process,** and **Serial** keys must match according to specific rules before an interest is said to match an event. (The matching rules are given later in this chapter.)

When a match is found, a copy of the event is distributed to the process that has the matching interest; the copy is placed on the local event queue of the process. A process' local event queue is a simple first-in, first-out queue. If a process has more than one matching interest, it receives one copy of the event for each matching interest.

This distribution procedure allows NeWS to broadcast the event that is at the head of its global event queue to many processes that are interested in the event. Each process that receives a copy of the event is given a chance to run before the next event is taken from the global event queue.

4. Reception of the event by processes with matching interests.

   To retrieve a delivered event from its local event queue, a process must execute the **awaitevent** operator. If an event is present on the process' local event queue, the **awaitevent** operator removes the event from the local queue and puts a copy of the event on the process' operand stack. The process can examine the keys in the event dictionary to determine what action it should take. If no event is waiting on the process' local event queue when the process executes **awaitevent**, the process blocks until an event is delivered.

3.2. Creating an Event

To create an event, use the **createevent** operator:

– **createevent** event

This operator creates an object of type *event* and places it on the top of the stack for the current process. Each of the object's keys has the value **null** (if the key is non-numeric) or zero (if the key is numeric).

The following example creates an event and associates it with a variable named e. The POSTSCRIPT language operators **begin** and **end** are then used to specify values for the **Name** and **Action** fields of the created event; each of these fields can take an arbitrary POSTSCRIPT language object as its value; the value is used to identify the event and match it to interests. Here, each value is specified as a string:

```
/e createevent def
e begin
   /Name (Hello) def
   /Action (There!) def
end
```

3.3. Expressing Interests

Before a process can receive an event, it must express an *interest* in receiving that type of event. To express an interest, use the **expressinterest** operator:

event **expressinterest** –
event process **expressinterest** –

The *event* argument can be an event created with **createevent** or it can be a system-generated event. If specified, the *process* argument indicates the process for which an interest is expressed; otherwise, an interest is expressed for the current process. An interest's type is still **eventtype**. Interests can be distinguished from other events by the **IsInterest** key; when an event is expressed as an interest, its **IsInterest** key is set to **true**.

If *event* is already an active interest, the call to **expressinterest** is ignored.

All events that match the *event* argument to **expressinterest** may be received by the specified process. The rules used to match events and interests are given in the Section 3.4, *Rules for Matching Events to Interests*.

Copying an Event Before Expressing Interest

Although events and interests use identical structures, NeWS does not allow you to dispatch into the event distribution mechanism an event that has already been expressed as an interest; nor does it allow you to express interest in an event that has been sent into the event distribution mechanism but not yet delivered.

When there is a danger of an event being used in this way, you can follow this procedure:

1. Create the event.
2. Make a copy of the event.
3. Express interest in the copy.
4. Send the event into distribution.

To make a copy of an event, use the POSTSCRIPT language **copy** primitive, as follows:

```
/e1 e createevent copy def
e1 expressinterest
```

Changing and Reusing Interests

The **Name** and **Action** key values of an interest can be changed after an interest has been expressed; the interest continues to be expressed and assumes the new **Name** and **Action** values that you have specified; these values are thus used in all future comparisons with distributed events.

Note, however, that none of the interest's other key values can be changed once the interest has been expressed; if you attempt to do this, an `invalidaccess` error is signaled and all the key values remain the same. To change any of the other key values, you must revoke the interest (using the **revokeinterest** operator) and then change and re-express the interest.

3.4. Rules for Matching Events to Interests

To determine whether an event matches an interest, NeWS examines the contents of the **Name**, **Action**, **Process**, and **Serial** dictionary keys. For each of these keys, the distributed event's value is compared to the interest's value; values are considered to match according to a special system of rules provided by NeWS. When all of the values match, the event and interest themselves match. This section summarizes the rules used to match events and interests.

Rules for Name And Action Key Matching

The **Name** and **Action** keys can contain values of any type. For an event to match an interest, the **Name** and **Action** keys must satisfy the following requirements:

- If the interest's key value is anything other than an array, a dictionary, or **null**, it must be identical to the event's key value.
- If the interest's key value is an array, at least one of its elements must be identical to the event's key value; if the key value is a dictionary, at least one of its keys must be identical to the event's key value.
- If the interest's key value is **null**, it matches anything in the key of the event.
- If the interest's key value is the name **AnyValue** or is an array or dictionary that contains **AnyValue**, it matches anything in the key of the event. (Note that if a dictionary contains both **AnyValue** and a value identical to the event's key value, the identical value is used.)
- If the event's key value is **null**, it matches only **null** or **AnyValue** in the corresponding key of the interest.

Rules for Process Key Matching

The **Process** key value of an event can be either a reference to a specific process or **null**. An interest's **Process** key value is never **null**; it is always automatically set to the process for which the interest is expressed. For an event to match an interest, the **Process** keys must satisfy the following rules:

- If the event's key value is **null**, it matches anything in the **Process** key of the interest.
- If the event's key value is a specific process, this value must be identical to the value of the interest's key.

Rules for Serial Key Matching

The **Serial** key of an event, which is read-only for both interests and events, is automatically set to a numeric value when the event is taken off the global event queue; the value is used to indicate the sequence in which the removal of events occurs. If the event is then successfully matched with an interest, the interest's **Serial** key is automatically set to the value that the event's key contains. NeWS allows an event to match an interest only when the interest's serial number is less than that of the event; this prevents an event passed to the **redistributeevent** operator from repeatedly matching the same interests before redistribution takes place.

NOTE *See Section* 3.12, *Using the Exclusivity Key, for a description of the* **redistributeevent** *operator. See Section* 3.8, *Using the Canvas Key: Matching Multiple Interests, for additional information on matching events and interests.*

3.5. Sending an Event into Distribution

When an event has been created, it can be sent into the NeWS event distribution mechanism. The mechanism contains a global event queue into which all sent events are immediately arranged according to the value of their **TimeStamp** key, which should be given a value by the process that creates the event (information on doing this is provided later in this chapter).

The server automatically removes each event from the front of the global event queue; an event at the head of the queue is removed when its **TimeStamp** value is less than or equal to the server's current time. When an event is removed, it is compared with the interests to locate matches. When a match is found, a copy of the event is distributed to the process that has the matching interest; the copy is placed on the local event queue of the process. Event copies remain in the local queue until the operator **awaitevent** is executed (see the following section for details).

To send an event, use the **sendevent** operator, whose syntax is as follows:

event **sendevent** –

In the following example, the previously defined event is sent into the event distribution mechanism. Since the value of the event's **TimeStamp** is zero by default, the event is immediately removed from the global event queue. A copy of the event is successfully matched to the interest previously expressed by the current process (e1 expressinterest). Therefore, a copy of the event is placed on the process' local event queue.

```
e sendevent
```

3.6. Awaiting Events

To retrieve the events that are queued on a process' local event queue, use the operator **awaitevent**.

– **awaitevent** event

When no event is contained on the process' local event queue, this operator causes the process to block; when an event arrives on the queue, **awaitevent** removes the event and places a copy of it on top of the process' operand stack; the process then ceases to block. If an event is waiting on the local queue when **awaitevent** is called, the event is immediately placed on the process' operand stack.

The following example executes **awaitevent** and then prints the values of the event's **Name** and **Action** keys.

```
awaitevent dup /Action get
exch /Name get
== ==
(Hello)
(There!)
```

3.7. Specifying the Name, Action, and Canvas Keys as Dictionaries

NeWS allows you to specify arrays or dictionaries as values for the **Name**, **Action**, and **Canvas** keys of an interest. For a process- or system-generated event to match the interest, each event key value must match one of the array or dictionary elements within the interest's corresponding key value. For example, suppose that the value of an interest's **Name** is a dictionary with three key-value pairs; for an event to match this interest, the event's **Name** key value must match one of the three keys in the interest's **Name** dictionary.

When an interest has a dictionary as one of its key values, the server performs some post-match processing on any event that matches the interest; the server handles the event differently depending on whether the interest has executable or non-executable values in its dictionary.

Non-Executable Dictionary Values

If the dictionary value associated with the matching key is non-executable, the value is automatically stored in the corresponding field of the event copy; that is, in the **Name**, **Action**, or **Canvas** field. The copy of the event is then placed on the top of the stack for the current process; thus, the new value can be retrieved.

This behavior is demonstrated by the following example. This example uses system-generated mouse button events. When the mouse button is pressed, the server generates an event that has the value of its **Name** key set to **/LeftMouseButton**, **/MiddleMouseButton**, or **/RightMouseButton**, depending on which mouse button is pressed; the value of the **Action** key is set to **/DownTransition**. When the mouse button is released, another event is generated with the same **Name** and with an **Action** of **/UpTransition**. Mouse events are described in detail in Section 3.9, *System-Generated Events*.

```
createevent dup begin                         % Create an event.
   /Name 3 dict dup begin                     % Create dict for the Name field.
      /LeftMouseButton (Left Button Went Down) def
      /MiddleMouseButton (Middle Button Went Down) def
      /RightMouseButton (Right Button Went Down) def
   end def
   /Action [ /DownTransition ] def            % Make Action be button presses.
   /Exclusivity true def
end dup expressinterest                       % Express interest in the event.

createevent dup begin                         % Create an event.
   /Name 3 dict dup begin                     % Create dict for the Name field.
      /LeftMouseButton (Left Button Went Up) def
      /MiddleMouseButton (Middle Button Went Up) def
      /RightMouseButton (Right Button Went Up) def
   end def
   /Action [ /UpTransition ] def              % Make Action be button releases.
   /Exclusivity true def
end dup expressinterest                       % Express interest in the event.

{
   awaitevent
   /Name get dup (Right Button Went Up) eq {
      == exit
   }{
      ==
   } ifelse
} loop

revokeinterest revokeinterest                 % Revoke both interests.
```

In this example, two interests are created: one interest in **/UpTransition** mouse button events and one interest in **/DownTransition** mouse button events. Each interest has a dictionary as the value of its **Name** key. Each **Name** dictionary contains three entries (one for each mouse button). Each entry has the **Name** of a mouse button event as the dictionary key and a string as the associated value; the string simply describes which button was pressed or released.

The **Exclusivity** key of each interest is set to true so that the interests are *exclusive*; an event that matches an exclusive interest is not compared to any other interests. Thus in this example, mouse presses and releases will not affect other canvases while these interests are expressed. For more information on exclusive interests, see Section 3.12, *Using the Exclusivity Key*.

After expressing these two interests, this example loops doing an **awaitevent**. When an event is retrieved from the process' local queue, the event's **Name** value is printed to the screen. If the event's **Name** value is (Right Button Went Up), the loop is exited.

Try typing this example to `psh` and then pressing the left and middle mouse buttons. Each time you press or release a mouse button, a message is printed to the screen. To exit the example, press and release the right mouse button.

Notice that for each matching button event, the string assigned in the interest's **Name** dictionary is substituted for the event's **Name** value before the event is distributed to the process. Thus, when the event's **Name** value is printed, the string is printed to the screen. For example, when the left mouse button is pressed, the string `(Left Button Went Down)` appears on the screen, instead of the name **/LeftMouseButton**.

Executable Dictionary Values

If the dictionary value associated with the matching key is executable, the corresponding event field is not modified; instead, the executable dictionary value is executed immediately after the received event is placed on the top of the stack by **awaitevent**. If more than one of the fields have executable values in their dictionaries, the **Name** value is executed first, followed by the **Action** value, followed by the **Canvas** value.

This behavior is demonstrated by the following example:

```
createevent dup begin                       % Create an event.
  /Name 3 dict dup begin                    % Create dict for the Name field.
  /LeftMouseButton {                        % event => –
    /Action get /UpTransition eq {
        (Left Button Up) ==
      } {
        (Left Button Down) ==
      } ifelse
    } def
    /MiddleMouseButton {                    % event => –
      /Action get /UpTransition eq {
        (Middle Button Up) ==
      } {
        (Middle Button Down) ==
      } ifelse
    } def
    /RightMouseButton {                     % event => –
      /Action get /UpTransition eq {
        (Right Button Up) ==
        exit
      } {
        (Right Button Down) ==
      } ifelse
    } def
  end def
  /Action [ /DownTransition /UpTransition ] def
  /Exclusivity true def
end dup expressinterest                     % Express interest in the event.

{
  awaitevent
} loop

revokeinterest
```

In this example, only one interest is expressed, but the interest contains both **/UpTransition** and **/DownTransition** in its **Action** field. Therefore, this interest can match both up and down mouse button events. Again, a dictionary is assigned to the interest's **Name** field. In this case, the dictionary values are executable; they are procedures that examine the **Action** of the event returned by **awaitevent** and then print the appropriate string. Each procedure also pops the event from the process' operand stack. Again, a release of the right mouse button causes an exit from the **awaitevent** loop.

When the mouse button is pressed or released, the server generates an event and distributes a copy of it to the process. After **awaitevent** places the event on the process' operand stack, the executable dictionary key value associated with the event's **Name** is executed immediately, printing the appropriate string to the screen.

NOTE *Executable matches, which are permitted by executable* **Canvas**, **Name**, *and* **Action** *dictionary values in interests, provide a highly efficient way of executing code according to the canvas on which an interest has been matched. Using this procedure, POSTSCRIPT language constructs such as* **case**, *which are normally used to vector a matched event to the correct handler, are made unnecessary.*

3.8. Using the Canvas Key: Matching Multiple Interests

The event dictionary contains a **Canvas** key, whose value can be one of the following:

- A canvas
- A dictionary or array that contains canvases
- The null value

The key has an important role to play in interest matching. This section describes the **Canvas** key and explains how multiple interests are matched according to its value. This section also describes the **EventsConsumed** key of the **canvastype** dictionary.

Pre-Child and Post-Child Interest Lists

NeWS provides each canvas with a *pre-child* and a *post-child* interest list. Interests are assigned to canvas interest lists as follows:

- When an interest is expressed with a canvas specified as its **Canvas** key value, the interest is inserted into one of the interest lists (pre-child or post-child) for the specified canvas.
- When an interest is expressed with an array or dictionary specified as its **Canvas** field, the interest is inserted into one of the interest lists for each canvas in the array or dictionary. One interest can therefore receive events sent to multiple canvases.

The event dictionary contains a key named **IsPreChild**. This key can be set in both events and interests, but is meaningless in events. When the key is set to **true** in an interest, it indicates that the interest appears on the *pre-child* interest list of the canvas. When the key is set to **false** , it indicates that the interest appears on the canvas' *post-child* interest list.

The kind of list (that is, pre-child or post-child) into which an interest is inserted determines the sequence in which events are matched across canvases. Generally speaking, the pre-child interests of a canvas' ancestors are matched before any of the interests of the canvas; the post-child interests of the ancestors are matched after the interests of the canvas. NeWS also provides ways in which the priority of interests can be specified and allows interests to become exclusive so that no other interest is matched after they themselves are matched.

A detailed analysis of interest lists and multiple event matching is provided in the following sections.

Order of Interest Matching

This section describes how multiple interests are matched across canvases.

When an event is distributed with **sendevent**, the value in the event's **Canvas** key determines which canvas interest lists are searched for potential matches. The exact search path through the canvas hierarchy depends on whether the **Canvas** key value of the event contains a single canvas, an array or dictionary containing multiple canvases, or null. These search paths are discussed in the following subsections:

Specifying a Single Canvas

When a single canvas is specified as an event's **Canvas** key value, the search procedure is as follows:

1. NeWS determines the branch of the canvas hierarchy that connects the specified canvas to the root canvas.
2. NeWS searches the pre-child interest list of each canvas on the branch, starting from the root canvas and ending with the specified canvas.
3. NeWS searches the post-child interest list of the specified canvas.

Therefore, when a single canvas is specified as an event's **Canvas** key value, the only post-child interest list to be searched is that of the specified canvas. This means that the event will not match post-child interests of the canvas' ancestors.

Specifying an Array or Dictionary

When an array or dictionary is specified as an event's **Canvas** key value, each element being a canvas, each canvas is considered in turn according to the rules described for a single canvas above.

NOTE *If the* **Canvas** *field of an interest contains a dictionary, it is subject to the same post-match rules as are the* **Name** *and* **Action** *fields. This allows* **Canvas** *field substitution and executable matches to occur.*

Specifying **null**

When **null** is specified as an event's **Canvas** key value, the *principal* canvas is the topmost canvas under the *x*, *y* location specified in the event. The search procedure is as follows:

1. NeWS determines the branch of the canvas hierarchy that connects the principal canvas to the root canvas.
2. NeWS searches the pre-child interest list of each canvas on the branch, starting from the root canvas and ending with the principal canvas.
3. NeWS searches the post-child interest list of each canvas on the branch, starting from the principal canvas and ending with the root canvas.

Therefore, when no canvas is specified as an event's **Canvas** key value, all pre- and post-child interest lists on canvases in the search path are searched.

NOTE *Any interest with* **null** *as its* **Canvas** *key value is on the pre-child interest list of the root canvas.*

Using the canvastype EventsConsumed Key

Although it is often desirable to affect a canvas' ancestors with operations that are intended to affect the canvas itself, it may sometimes be necessary to override the procedure you have defined to allow this. The **canvastype** dictionary contains a key named **EventsConsumed**; this key allows you to specify whether events tested for a match with the current canvas' post-child interests are similarly tested with the post-child interests of the canvas' parent; the pre-child interests of the canvas' parent are always tested. The possible values for the **EventsConsumed** key are as follows:

- **/AllEvents**

  This indicates that all events tested for a match with the canvas' post-child interests are consumed; that is, none is tested for a match with the post-child interests of the canvas' parent.

- **/MatchedEvents**

  This indicates that events successfully matched with one or more of the canvas' post-child interests are consumed; that is, they are not tested for a match with the post-child interests of the canvas' parent. However, events not successfully matched with the canvas' post-child interests will indeed be tested against the post-child interests of the canvas' parent.

  /MatchedEvents is the default for the **EventsConsumed** key of all canvases.

- **/NoEvents**

  This indicates that no events tested for a match with the canvas' post-child interests are consumed; that is, all are tested against the post-child interests of the canvas' parent.

Non-consumed events are tested against the post-child interests of the canvas' grandparent depending on the **EventsConsumed** status of the canvas' parent. Thus, if all canvases in a branch extending to the root canvas have **/NoEvents** specified, all events are tested against all post-child interests of each canvas.

For each successful match that occurs, the server places one copy of the current distributed event on the local event queue for the process that has the matching interest. When **awaitevent** is called, the event is placed on the process' stack.

NOTE *Each process in NeWS maintains an interest list. The list contains all interests currently expressed by the process. For further information see Section* 3.14, *Using the Process Key.*

Multiple Post-Child Interest Matching: An Example

The following example shows how multiple post-child interests can be matched.

```
/MakeFirstCanvas {                              % Make a parent canvas.
   /FirstCanvas framebuffer newcanvas def
   newpath 0 0 250 250 rectpath
   FirstCanvas reshapecanvas
   FirstCanvas setcanvas
   1 fillcanvas
   FirstCanvas /Mapped true put
   25 25 movecanvas
   newpath 3 3 244 244 rectpath
   clipcanvas
   1 fillcanvas
} def

/MakeSecondCanvas {                              % Make a child canvas.
   /SecondCanvas FirstCanvas newcanvas def
   newpath 0 0 75 75 rectpath
   SecondCanvas reshapecanvas
   SecondCanvas /Transparent false put
   SecondCanvas setcanvas
   25 25 movecanvas
   1 fillcanvas
   SecondCanvas /Mapped true put
   newpath 3 3 69 69 rectpath
   clipcanvas
   0 fillcanvas
} def

/Flush {                                         % Clear the process event queue.
   countinputqueue {awaitevent pop} repeat
} def

/MakeEvent {                                     % Make an event object.
   /e1 createevent def
   e1 begin
      /Name /RightMouseButton def
      /Action /DownTransition def
   end
} def

/MakeInterests {                                 % Make event objects, specific
   /IntX e1 createevent copy def                 % to each canvas, in which
   IntX /Canvas SecondCanvas put                 % interest can be expressed.
   /IntY e1 createevent copy def
   IntY /Canvas FirstCanvas put
} def

/ChildOp {
   FirstCanvas setcanvas .8 fillcanvas
   SecondCanvas setcanvas .8 fillcanvas
   SecondCanvas /EventsConsumed /NoEvents put
   IntX expressinterest                          % Set the EventsConsume
   IntY expressinterest                          % key of SecondCanvas
   Flush                                         % to /NoEvents, express
   /Count 0 def                                  % interests, send the event,
   2{                                            % change canvas appearances
      awaitevent                                 % accordingly, and revoke
      /Interest get dup                          % interests.
      /Canvas get dup
      setcanvas Count fillcanvas
      /Count Count .1 add def
      Count 1 gt {/Count 0 def} if
    } repeat
    IntX revokeinterest
    IntY revokeinterest
    (ChildOp has completed.\n) print
} def

MakeFirstCanvas                               % Call all operations.
MakeSecondCanvas
MakeEvent
MakeInterests
ChildOp
                                              % Click the right mouse button
                                              % over the position of
                                              % SecondCanvas.
```

This example begins by creating two canvases, named FirstCanvas and SecondCanvas. SecondCanvas is the child of FirstCanvas; both canvases have **EventsConsumed** set to **/MatchedEvents** by default.

The Flush operation, which is used to clear the process event queue, contains the following primitive:

– countinputqueue num
The operator returns the number of events currently available from the process' local event queue.

The Flush operation retrieves the number of queued events, calls **awaitevent** on each of them, and removes each of them from the process stack.

The operation MakeEvent is used to create an event object, named e1, whose **Name** is **/RightMouseButton** and whose **Action** is **/DownTransition**. This event object will be used to derive canvas-specific interest objects in which interest may be expressed. Note that the distributed event corresponding to these interests will be a system-generated event produced by pressing the right mouse button. System-generated events, which do not require calls to **sendevent**, are explained in Section 3.9, *System-Generated Events*.

The MakeInterests operation creates two interest objects that match the previously created event; each specifies one of the two defined canvases. The **IsPreChild** key is not set for either interest object; thus, the interests are both *post-child* by default.

The ChildOp operation establishes the **EventsConsumed** key value of SecondCanvas as **NoEvents**; thus, events that occur directly over SecondCanvas will be compared with interests expressed by both SecondCanvas and FirstCanvas.

Having expressed interest in the previously defined interest objects, ChildOp makes two iterative calls to **awaitevent**. The operation specifies that whenever an event appears on the local event queue, the name of the canvas that has expressed the corresponding interest will be derived; this canvas will then be established as the current canvas, and its color will be modified by a call to **fillcanvas**; the color value being derived from the Count variable, incremented by 0.1 each iteration. Thus, the canvas whose interests are first matched is changed to the color closer to zero (that is, the darker color).

Therefore, if the mouse is placed over SecondCanvas, and the right mouse button is clicked, an event identical to e1 is automatically sent. The event is compared with interests owned by SecondCanvas, and a match is made. Since SecondCanvas is consuming no events, the event is then compared with interests owned by FirstCanvas, and another match is made. Two copies of e1 are thus placed on the local event queue of the current process.

When **awaitevent** is called the first time, it causes the color of SecondCanvas to be modified to the initial value of the Count variable, which is zero. When **awaitevent** is called the second time, it causes the color of FirstCanvas to be modified to the second value of Count, which is 1. Thus, SecondCanvas appears darker than FirstCanvas: this is shown by the following illustrations, which represent the appearance of the canvases before and after the mouse button is clicked.

Figure 3-1 *Initial Appearance of Canvases*

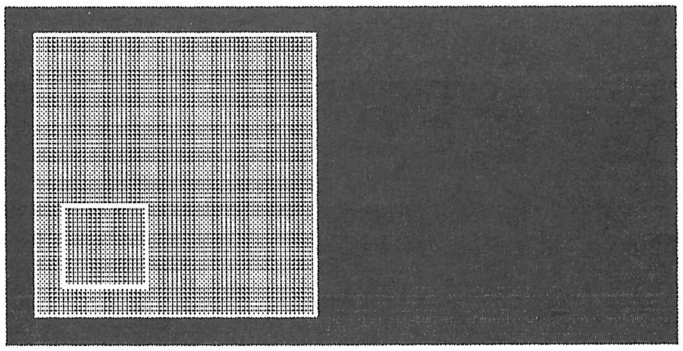
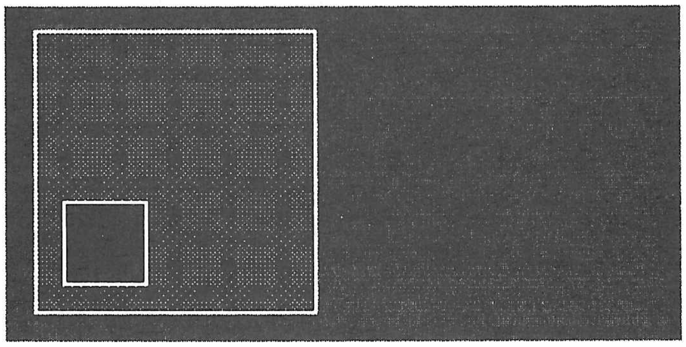


Figure 3-2 *Result of Pre-Child Interest Matching*



Note that ChildOp also calls the following primitive:

event **revokeinterest** –
event process **revokeinterest** –
The *event* argument specifies an event in which interest has previously been expressed. The optional *process* argument specifies the process on whose behalf

the interest is revoked; if no process is specified, interest is revoked on behalf of the current process.

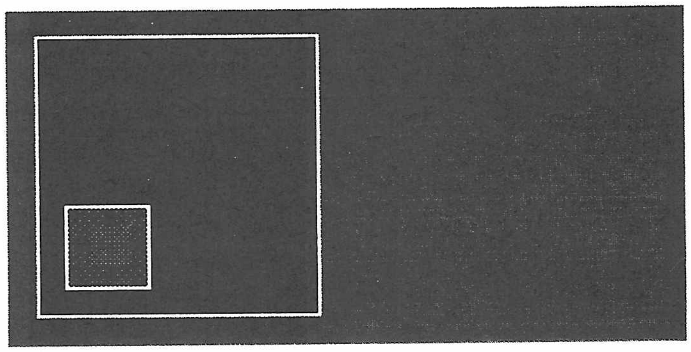
Multiple Pre-Child Interest Matching: An Example

The following example modifies the **IsPreChild** key values of the previously created interest objects:

```
IntX /IsPreChild true put           % Change interest list status of
IntY /IsPreChild true put           % existing interests.

ChildOp
                                    % Click the right mouse button over
                                    % the position of SecondCanvas.
```

In this example, since the values of the **IsPreChild** keys are set to **true** , calling ChildOp causes the pre-child interest of FirstCanvas to be matched before the pre-child interest of SecondCanvas; thus, FirstCanvas appears darker than SecondCanvas, as shown by the following illustration:

Figure 3-3 *Result of Multiple Pre-Child Interest Matching*



3.9. System-Generated Events

A *system-generated* event is created and sent automatically by NeWS in the following circumstances:

- The mouse is manipulated.
- A keyboard-key is pressed.
- A canvas is damaged.
- An object becomes obsolete, and its memory needs reclaiming.
- A process dies while it is still referenced.
- The mouse pointer exits one canvas and enters another.

Since these events are created and sent automatically, the primitives **createevent** and **sendevent** do not need to be used; however, the other NeWS primitives for expressing interest and awaiting events must be used in the same way as is required for process-generated events.

This section describes system-generated events and shows how they can be used.

Mouse Events

NeWS automatically generates events that correspond to the status of the mouse. Each event has an appropriate value automatically inserted in its **Name** and **Action** key. Events are generated in the following circumstances:

- The mouse is moved.

  The value of the **Name** key is set to **/MouseDragged**; the value of the **Action** key is set to **null.**

- A mouse button is pressed and released.

  When the mouse button is pressed, the value of the **Name** key is set to **/LeftMouseButton, /MiddleMouseButton**, or **/RightMouseButton**, depending on which button is pressed; the value of the **Action** key is set to **/DownTransition**. When the button is released, another event is generated with the same **Name** value and with the **Action** set to **/UpTransition**. Thus, two events are automatically generated whenever a mouse button is pressed and released.

The following example demonstrates mouse button events:

```
%
% Create canvas to play in.
%
/canvas framebuffer newcanvas def        % Create a canvas object.
100 100 translate                        % Move its origin.
0 0 400 400 rectpath                     % Make a rectangular path.
canvas reshapecanvas                     % Make our canvas that shape.
canvas /Mapped true put                  % Map the canvas.
canvas setcanvas                         % Make canvas the currentcanvas.
1 fillcanvas                             % Give it a white background.
0 setgray                                % Draw with black lines.

%
% Print (in the canvas) documentation
% on button usage
%
/Times-Roman findfont 12 scalefont setfont
10 30 moveto
(Press left button to move currentpoint) show
10 20 moveto
(Press middle button and drag to draw a line) show
10 10 moveto
(Press right button to quit) show
200 200 moveto                           % set starting point.

%
% Create an interest in MouseDragged events on our play canvas
% (store in /drag); this is an executable match that draws a
% line to the current mouse position each time the mouse moves
% while this interest is expressed. It also leaves the
% currentpoint at the mouse position.
%
/drag createevent dup begin
```

```
   /Name 1 dict dup begin
      /MouseDragged {                          % event => –
         begin
            XLocation YLocation lineto stroke  % Consumes the path.
            XLocation YLocation moveto         % Set currentpoint to same.
         end
      } def
   end def
   /Action null def
   /Canvas canvas def
end def

%
% Create an interest in Up and Down transitions of all
% three mouse buttons. Each button has its own handler
% associated with it by the value of the corresponding key
% in the /Name field of the interest.
%
createevent dup begin
   /Name 3 dict dup begin
      /LeftMouseButton {                       % event => –
         begin
            XLocation YLocation moveto         % Move the currentpoint.
         end
      } def
      /MiddleMouseButton {                     % event => –
         begin
            Action /DownTransition eq {
               drag expressinterest            % We want drag events now.
               XLocation YLocation lineto stroke % Stroke consumes the path.
               XLocation YLocation moveto      % So set currentpoint back.
            } {
               drag revokeinterest             % Don't want drag events any more.
            } ifelse
         end
      } def
      /RightMouseButton {                      % event => –
         pop                                   % We're all done...
         exit                                  % Break out of the {} loop.
      } def
   end def
   /Action [ /DownTransition /UpTransition ] def
   /Canvas canvas def
end dup expressinterest

{ awaitevent } loop                            % Loop, processing events.

revokeinterest
canvas /Mapped false put                       % Unmap the window.
/canvas null def                               % Free the memory.
```

This example creates a canvas and maps it to the screen. It then prints three strings to the canvas to provide user instructions for the example. After preparing the canvas, an interest named drag is created for **/MouseDragged** events. The interest uses an executable value in the **Name** dictionary; the procedure strokes a line to the *x*, *y* location of the event and then sets the current point to be the endpoint of the line. This interest is not expressed immediately.

Another interest is then created; the second interest is for mouse button presses and releases. This interest also uses executable values in its **Name** dictionary. When a left mouse button event is matched, a procedure moves the current point to the *x*, *y* location of that event. When a middle mouse button event is matched, a procedure checks to see if the event is a **/DownTransition.** If so, drag is passed to **expressinterest**. The drag interest is revoked when the button is released. When a right mouse button event is matched, a procedure pops the event and exits the **awaitevent** loop.

Try running this example with `psh` and drawing in the canvas that is generated.

Enter and Exit Events

NeWS generates a special event whenever the cursor crosses the boundary of a canvas. The **Name** and **Action** key values of the event are automatically set according to the kind of movement that has occurred and the relationship between the canvases concerned.

Name Key Values

The value of the **Name** key is automatically set to either **ExitEvent** or **EnterEvent**, depending on the movement of the mouse. Thus, when the mouse crosses any canvas boundary, at least two events are generated; the first event is the exit event for the canvas being exited; the second event is the enter event for the canvas being entered.

The following example demonstrates the use of **EnterEvents**:

```
/EntryOp {
   /e1 createevent def                     % Create an entry event
      e1 begin                             % for FirstCanvas.
         /Name /EnterEvent def
         /Canvas FirstCanvas def
   end
   /e2 createevent def                     % Create an entry event
      e2 begin                             % for SecondCanvas.
         /Name /EnterEvent def
         /Canvas SecondCanvas def
      end
      e1 expressinterest                   % Express interests.
      e2 expressinterest
      Flush
      /Toggle 0 def
      10 {
         awaitevent dup /Name get
         /EnterEvent eq {
            /Interest get dup              % Modify canvas colors
            /Canvas get setcanvas          % whenever an entry
            Toggle fillcanvas} if          % event occurs.
```

```
                clear
                Toggle dup {
                    0 {/Toggle .5 def}
                    .5 {/Toggle 1 def}
                    1 {/Toggle 0 def}
                } case
            } repeat
            e1 revokeinterest
            e2 revokeinterest
            (EntryOp has completed.\n) print
    } def

    SecondCanvas setcanvas
    0 fillcanvas
    FirstCanvas setcanvas
    1 fillcanvas
    EntryOp
```

In this example, both SecondCanvas and FirstCanvas are specified to change color when an **EnterEvent** occurs. The following illustrations respectively show the appearance of the canvases when the mouse enters FirstCanvas from the framebuffer, enters SecondCanvas from FirstCanvas, and re-enters FirstCanvas from SecondCanvas:

Figure 3-4 *Initial Appearance of FirstCanvas and SecondCanvas*

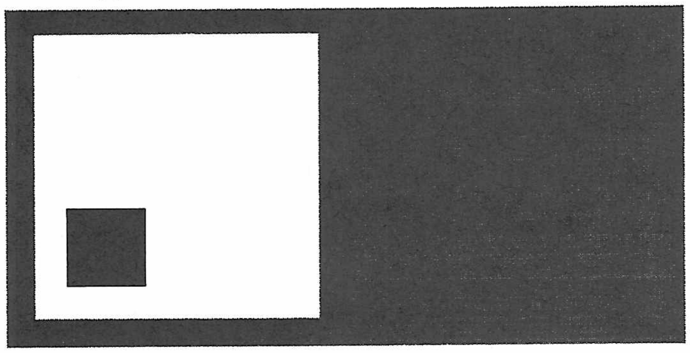


Figure 3-5 *First Entry Event, Matched by FirstCanvas*

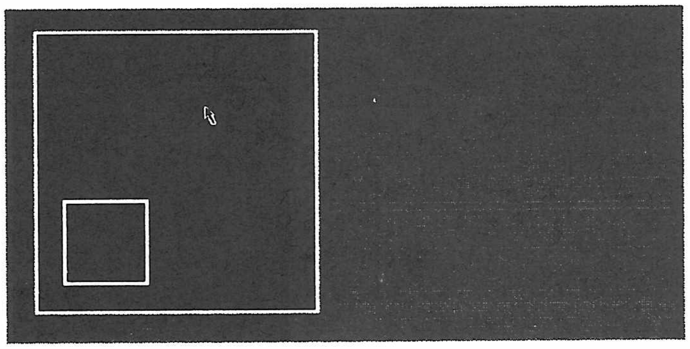


Figure 3-6 *Second Entry Event, Matched by SecondCanvas*

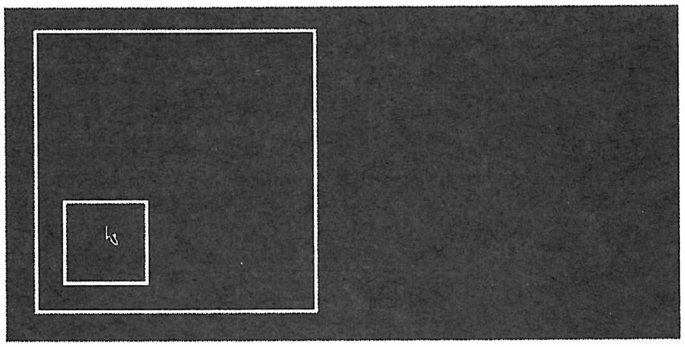
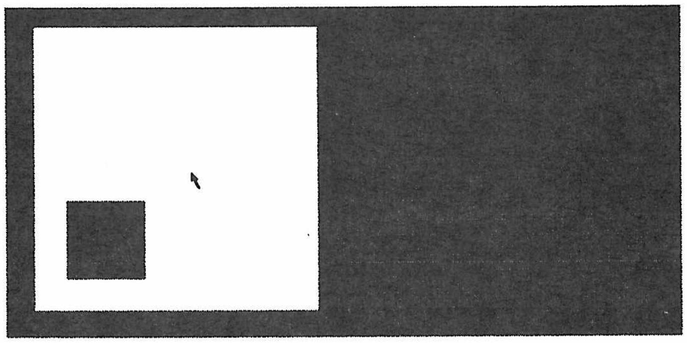


Figure 3-7 *Third Entry Event, Matched by FirstCanvas*



Action Key Values

The value of the **Action** key is automatically set to a numeric value that corresponds to the movement of the mouse and the relationship between the canvases between which it moves.

The following table describes each numeric value to which **Action** is set. Note that a canvas is said to contain the cursor *directly* when it is the frontmost canvas under the mouse; a canvas is said to contain the cursor *indirectly* if it is an ancestor of a canvas that *directly* contains the mouse. Note also that a canvas does not receive a crossing event if it contains the cursor directly both before and after the cursor movement; nor does it receive a crossing event if it contains the cursor indirectly both before and after the cursor movement.

Table 3-1 *Boundary Crossing Events*

Name	*Action*	*Explanation*
/EnterEvent	0	The canvas now *directly* contains the cursor; the previous direct container was an ancestor of this canvas.
	1	The canvas now *indirectly* contains the cursor; the previous direct container was an ancestor of this canvas.

Table 3-1 *Boundary Crossing Events— Continued*

Name	*Action*	*Explanation*
	2	The canvas now *directly* contains the cursor; the previous direct container was a descendant of this canvas.
	3	The canvas now *directly* contains the cursor; the previous direct container was not an ancestor or descendant of this canvas.
	4	The canvas now *indirectly* contains the cursor; the previous direct container was not an ancestor or descendant of this canvas.
/ExitEvent	0	The canvas formerly contained the cursor *directly*; the new direct container is an ancestor of this canvas.
	1	The canvas formerly contained the cursor *indirectly*; the new direct container is an ancestor of this canvas.
	2	The canvas formerly contained the cursor *directly*; the new direct container is a descendant of this canvas.
	3	The canvas formerly contained the cursor *directly*; the new direct container is not an ancestor or descendant of this canvas.
	4	The canvas formerly contained the cursor *indirectly*; the new direct container is not an ancestor or descendant of this canvas.

Using the **postcrossings** Operator

The **postcrossings** operator generates *canvas crossing events*, which notify the system of the movement from one canvas to another of a *state*, such as the canvas under the pointer or the focus. Examples of crossing events are **Enter** events, **Exit** events, and focus notification events (explained in *Focus Events*, below). The **Action** field values of the crossing events comply with X11 focus and enter/exit event specification.

See Chapter 9, *NeWS Operator Extensions*, for a complete description of the **postcrossings** operator.

Using the XLocation and YLocation Keys

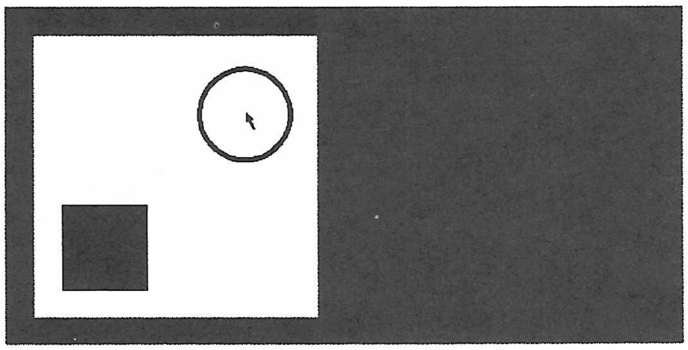
The **eventtype** dictionary contains **XLocation** and **YLocation** keys, which respectively contain the *x* and *y* coordinates at which the event occurred. These keys are arbitrary in process-generated events and interests; their values are automatically set in system-generated events. Events coordinates are reported with respect to the current transformation matrix.

The following example demonstrates how the **XLocation** and **YLocation** keys can be used:

```
FirstCanvas setcanvas
1 fillcanvas

/e1 createevent def

e1 begin
   /Name /LeftMouseButton def
   /Action /DownTransition def
   /Canvas FirstCanvas def
end

/DrawCircle {
   4 setlinewidth
   e1 expressinterest
   Flush
   0 setgray
   awaitevent dup /XLocation get
   exch dup /YLocation get
   exch pop
   40 0 360 arc
   stroke
   e1 revokeinterest
} def

DrawCircle                                % Click the left mouse button.
```

The above example requires that the operation DrawCircle be called and the left mouse button clicked over FirstCanvas. The **XLocation** and **Ylocation** values for the corresponding event are retrieved and used in a call to the **arc** operation, which draws a circle centered on the retrieved coordinates.

Figure 3-8 *Result of Mouse-Generated Event*



Using the Coordinates Key

The **eventtype** object contains a **Coordinates** key, which provides a way to get and set the *x,y* location of an event atomically. The field accepts an array of length two, with the *x* coordinate in the first position and the *y* coordinate in the second.

Focus Events

Focus events are generated by the NeWS *focus manager* through the *postcrossings mechanism*. These events signal a change in the *focal point* of the keyboard, which determines the canvas that is to receive keyboard input. The **Name** value of a focus event is always **/RestoreFocus**, **/AcceptFocus**, or **/LoseFocus**. The **Action** value is an integer specifying the nature of the focal change. These integers and their significance are shown by the following table:

Table 3-2 *Input Focus*

Name	*Action*	*Explanation*
/RestoreFocus **/AcceptFocus**	0	The canvas is now the focus; the previous focus was an ancestor of this canvas.
	1	The canvas is now the ancestor of the focus; the previous focus was an ancestor of this canvas.
	2	The canvas is now the focus; the previous focus was a descendant of this focus.
	3	The canvas is now the focus; the previous focus was not an ancestor or descendant of this canvas.
	4	The canvas is now an ancestor of the focus; the previous focus was not an ancestor or descendant of this canvas.
	5	The canvas directly or indirectly contains the pointer and is now a descendant of the focus. The previous canvas is not equivalent to this canvas nor is the previous canvas an ancestor or descendant of this canvas.

Table 3-2 *Input Focus— Continued*

Name	*Action*	*Explanation*
	6	The focus is now **ReDistribute**.
	7	The focus is now **None**.
/LoseFocus	0	The canvas used to be the focus; the new focus is an ancestor of this canvas.
	1	The canvas used to be an ancestor of the focus; the new focus is an ancestor of this canvas.
	2	The canvas used to be the focus; the new focus is a descendant of this canvas.
	3	The canvas used to be the focus; the new focus is not an ancestor or descendant of this canvas.
	4	The canvas used to be an ancestor of the focus; the new focus is not an ancestor or descendant of this canvas.
	5	The canvas directly or indirectly contains the pointer and used to be a descendant of the focus. The new canvas is not equivalent to this canvas nor is the new canvas an ancestor or descendant of this canvas.
	6	The focus used to be **ReDistribute**.
	7	The focus used to be **None**.

Keyboard Events

Keyboard events are generated in response to the user's pressing a key on the keyboard. These events have a **Name** value that is a number in the range of 28416 to 28671 (6F00 to 6FFF hexidecimal) and an **Action** value of **/UpTransition** or **/DownTransition**. The name of the keyboard event does not represent the character that is encoded on the key; it represents an implementation-dependent keyboard encoding.

The Repeat Key Dictionary

One of the dictionaries within **systemdict** is named **repeatkeydict**. This is a dictionary specifying which keyboard keys should repeat. All of the key codes defined in the dictionary are eligible for repeating. Note that the user should not manipulate this dictionary directly but should use **ClassRepeatKeys**.

ClassRepeatKeys controls the key repeating characteristics of the server; its class methods are described below. For a basic explanation of classes and class methods, see Chapter 4, *Classes*.

– **/interval** num
num **/setinterval** –
Get and set the keyboard repeat interval. *num* specifies how fast the keyboard will repeat and is in units of 2^{16} milliseconds.

– **/threshold** num
num **/setthreshold** –
Get and set the keyboard repeat threshold. *num* specifies the amount of time a key must be depressed before it will begin to repeat. *num* is in units of 2^{16} milliseconds.

– **/repetition** boolean
boolean **/setrepetition** –
Get and set the global state of repeat keys. If *boolean* is true, the keyboard will repeat.

keycode **/inhibitrepeat** –
keycode **/allowrepeat** –
Allow or inhibit the repeating of a particular key.

The following values can be specified in the user's **UserProfile** (in `.startup.ps`):

/KeyRepeatThresh
The initial repeat threshold in units of 2^{16} milliseconds.

/KeyRepeatTime
The initial repeat interval in units of 2^{16} milliseconds.

Damage Events

Damage events are generated for a canvas whenever it is *damaged* (a definition of *damage* is provided in Chapter 2, *Canvases*). The server will not send another event until the damage has been cleared by use of the **damagepath** operator. The **Action** key value for the event is null; the **Canvas** key value specifies the affected canvas.

Obsolescence Events

Obsolescence events are generated by the server for an object that becomes obsolete. *Obsolescence* is defined as the state where all the references to an object are *soft*. (See the discussion of *soft references* in Chapter 7, *Memory Management*). The **Name** field of the event is **/Obsolete**; the **Action** field is the obsolete object.

ProcessDied Events

ProcessDied events are generated when a lightweight process dies. The **Name** key value of the event is **/ProcessDied**; the **Action** key value is the process itself. Note that no **ProcessDied** event is generated if the process dies when no references to it exist or no **waitprocess** is being executed upon it.

3.10. Using the ClientData Key

The **eventtype** object contains a **ClientData** key. The value of this key may be set to any NeWS object; the object can be accessed at any time. Although new keys may be added to an event dictionary, doing so adds memory overhead. The **ClientData** key is useful if the programmer has only one piece of information to add to the event dictionary.

The following example demonstrates how the **ClientData** key can be used:

```
FirstCanvas setcanvas
1 fillcanvas
Flush

/e1 createevent def
e1 begin
   /Name /Hello def
   /ClientData {.2 fillcanvas} def
end

/ex e1 createevent copy def

/ClientDataOp {
   e1 expressinterest
   ex sendevent
   awaitevent dup /Name get
   /Hello eq { /ClientData get exec } if
   e1 revokeinterest
} def

ClientDataOp
```

In the above example, the value of the **ClientData** key for e1 is specified as a call to **fillcanvas**. When a successful match has been made between the event and a corresponding interest, the value of the key is retrieved and executed.

3.11. Using the Priority Key

The interests contained in the interest list of any canvas can be assigned different *priorities*. The interest that has the highest priority is always the first interest in the list with which a distributed event is compared and may thus be the first interest matched. The interest that has the lowest priority is always the last with which the event is compared.

The **eventtype** dictionary includes a **Priority** key that allows you to specify a priority; note that a specified priority is meaningless when the event object is used as an event rather than an interest. The **Priority** key value can be set to any number; the default value is 0; negative and fractional values are permitted. The highest number signifies the highest priority. When interests have the same priority (which is the default), exclusive interests are compared first. Among non-exclusive interests of the same priority, the most recently expressed interest is compared first.

The following example demonstrates use of the **Priority** key:

```
FirstCanvas setcanvas
0 fillcanvas
Flush

/e1 createevent def
e1 begin
   /Name /Hello def
end

/IntA e1 createevent copy def
IntA begin
   /ClientData {.04 sleep .8 fillcanvas} def
   /Canvas FirstCanvas def
   /Priority 1 def                           % The canvas goes light gray
end                                          % when interest IntA is matched.

/IntB e1 createevent copy def
IntB begin
   /ClientData {.04 sleep .2 fillcanvas} def
   /Canvas FirstCanvas def
   /Priority 0 def                           % The canvas goes dark gray
end                                          % when interest IntB is matched.

/e1a e1 createevent copy def
e1a begin
   /Canvas FirstCanvas def
end

/ReceiveEvents {                             % Retrieve and execute the
   2 {                                       % value of the matched interest's
      awaitevent /Interest get dup           % ClientData key.
      /ClientData get exec
   } repeat
} def

/Waiting {
   IntA expressinterest
   IntB expressinterest
   e1a sendevent
   ReceiveEvents
   IntA revokeinterest
   IntB revokeinterest
} def

Waiting
```

The above example creates an event that is matched by two post-child interests on a single canvas. Initially, the interests have **Priority** values of 1 and 0 respectively; thus, when the event is sent, the interest with **Priority** 1 is matched first.

Two calls to **awaitevent** are then made and the corresponding events are placed on the process' stack.

The **ClientData** key for each interest contains a call to the **fillcanvas** primitive, preceded by a call to **sleep**. When the interest IntA is matched, the current canvas turns light gray; when IntB is matched, the canvas turns dark gray. Thus, since the priority of IntA is higher than that of IntB, the canvas turns light gray first, then dark gray.

In the following example, the respective priorities of the interests are reversed; thus, the order of color changes made to the current canvas is also reversed.

```
0 fillcanvas
Flush
IntB /Priority 1 put
IntA /Priority 0 put
Waiting
```

3.12. Using the Exclusivity Key

The **eventtype** dictionary contains an **Exclusivity** key. This key is significant only for interests; its value is ignored in distributed events. The value of the key can be set to either **true** or **false**: if the value is **true**, a distributed event successfully matched with this interest is not compared with any further interests. Note that the **Exclusivity** key prohibits interest-comparison across all processes and all canvases.

The following example (which modifies the code used in Section 3.11, *Using the Priority Key*) demonstrates how the **Exclusivity** key can be used:

```
0 fillcanvas
Flush
IntB /Exclusivity true put

/NewReceiveEvents {
   2 {
   awaitevent /Interest get dup
   /ClientData get exec
   dup /Exclusivity get {exit} if
   } repeat
} def

/NewWaiting {
   IntA expressinterest
   IntB expressinterest
   e1a sendevent
   NewReceiveEvents
   IntA revokeinterest
   IntB revokeinterest
   (NewWaiting has completed.\n) print
} def

NewWaiting
```

In the above example, the **Exclusivity** key of IntB is set to **true**. Thus, since the **Priority** of IntB is currently higher than that of IntA, IntA is not matched following the successful match made with IntB.

Using the redistributeevent Operator

The **redistributeevent** operator effectively allows you to override the exclusivity of an interest. The operator is as follows:

event **redistributeevent** –
The *event* argument should be an event already returned by **awaitevent**; the **redistributeevent** operator continues the distribution process, comparing the specified event with all available interests, starting from the interest immediately after the successfully matched interest that permitted the event object to be returned by **awaitevent**. Note that **redistributeevent** does not reinsert the event into the global event queue. No interest compared with the specified event since the last call to **sendevent** is compared with that event again.

The following example, which modifies the previous example, shows how **redistributeevent** can be used:

```
/NewReceiveEvents {
   2 {
   awaitevent /Interest get dup
   /ClientData get exec
   dup /Exclusivity get {e1a redistributeevent} if
   } repeat
} def

0 fillcanvas
NewWaiting
```

In the above example, **redistributeevent** is called when the **Exclusivity** key of the matched interest is determined to be **true**.

3.13. Using the TimeStamp Key

The **eventtype** dictionary contains a **TimeStamp** key, whose value indicates the time after which the event may be removed from the global event queue for comparisons to be made with available interests. Time values are measured in units of 2^{16} milliseconds. No event can be removed from the queue before its **TimeStamp** value signifies the current time; thus, when an event contains a **TimeStamp** value that specfies a time in the future, the event must — following the call to **sendevent** — remain in the global event queue until the appropriate time is reached.

The following example demonstrates how the **TimeStamp** value can be used:

```
FirstCanvas setcanvas
1 fillcanvas
Flush

/PrepareEvents {
   /e1 createevent def
   e1 begin
      /Name /Dark def
   end
   /e2 createevent def
   e2 begin
      /Name /Medium def
   end
   /e3 createevent def
   e3 begin
      /Name /Light def
   end
   /ex e1 createevent copy def
   /ey e2 createevent copy def
   /ez e3 createevent copy def
   ex /TimeStamp currenttime .07629 add put
   ey /TimeStamp currenttime .15259 add put
   ez /TimeStamp currenttime .30518 add put
   e1 expressinterest
   e2 expressinterest
   e3 expressinterest
```

```
    ez sendevent
    ey sendevent
    ex sendevent
} def

/Transform {
    3 {
        awaitevent /Name get
        dup {
            /Dark {.2 fillcanvas}
            /Medium {.5 fillcanvas}
            /Light {.8 fillcanvas}
        } case
    } repeat
    e1 revokeinterest
    e2 revokeinterest
    e3 revokeinterest
    (Transform has completed.\n) print
} def

PrepareEvents
Transform
```

In the above example, the operation PrepareEvents is used to create and express interest in three events, each with its own **Name** and **TimeStamp** value. Note that each **TimeStamp** value is specified as a value added to the value of **currenttime** when the event is sent; thus, each event remains in the global event queue until its specified **TimeStamp** value becomes the current time.

The operation Transform is used to call **awaitevent** and examine the **Name** value of each event returned to the local event queue; the color of the current canvas is then changed in accordance with the **Name** value. Note that the sequence of color changes indicates that the events were matched and processed in the reverse order of their sending (that is, in the correct order as specified by their respective **TimeStamp** values).

Using the recallevent Operator

NeWS provides a **recallevent** operator that allows you to recall an event that has not yet been distributed. The operator is as follows:

event **recallevent** –

The *event* argument should be an event object that is currently in the global event queue.

The following example (which modifies the previous example) shows how **recallevent** can be used:

```
1 fillcanvas
Flush

/NewTransform {
    3 {
            awaitevent /Name get
            dup {
               /Dark {.2 fillcanvas}
               /Medium {.5 fillcanvas
                     ez recallevent
                     ex /TimeStamp currenttime .15259 add put
                     ex sendevent}
               /Light {.8 fillcanvas}
            } case
         } repeat
         e1 revokeinterest
         e2 revokeinterest
         e3 revokeinterest
      (NewTransform has completed.\n) print
      } def

      PrepareEvents
      NewTransform
```

The above example redefines the operation NewTransform so that the event ez is recalled and ex is resent with a new **TimeStamp** value.

3.14. Using the Process Key

The **eventtype** dictionary contains a **Process** key, which can be used to specify a process. If a distributed event specifies a process in this way, the event is compared only with interests expressed by that process. The default value for this key is **null**, which means that the event can be compared with interests expressed by any process.

The **Process** key value of an interest is always automatically set to the process in which the interest is expressed.

The following example shows how the **Process** key can be used:

```
/ProcessProg {
   /ParentProc{
      FirstCanvas setcanvas
      /e1 createevent def
         e1 begin
            /Name /Hello def
         end
      /e2 e1 createevent copy def
      e2 /Process ParentProc put
      /IntZ e1 createevent copy def
      IntZ expressinterest
      e2 sendevent
      e1 sendevent
            {
            SecondCanvas setcanvas
            /IntX e1 createevent copy def
            IntX expressinterest
            awaitevent /Process get ==
            .5 fillcanvas
```

```
                IntX revokeinterest
                } fork waitprocess
            awaitevent /Process get ==
            .5 fillcanvas
            IntZ revokeinterest
        } fork def
    } def

    FirstCanvas setcanvas
    1 fillcanvas
    SecondCanvas setcanvas
    0 fillcanvas
    ProcessProg
```

The above example defines a parent process (named ParentProc), and an unnamed child process. The parent process sends two events; the first has the name ParentProc specified as its **Process** key value and is thus acceptable only to interests generated by the parent process itself; the other has **null** as its **Process** key value and can thus be accepted by any process, specifically by the child process that ParentProc generates.

3.15. Input Synchronization with Multiple Processes

NeWS synchronizes the event distribution process so that when an event is removed from the front of the global event queue, is successfully matched with one or more interests, and has copies of itself placed on the local event queues of the relevant processes, no other event is removed from the global queue until each of the relevant processes has had a chance to run. Similarly, when an event is passed to **redistributeevent**, NeWS will not remove an event from the queue until processes that receive the redistributed event have either completed or blocked.

Using **blockinputqueue**

In interactive window management, event distribution must often be explicitly synchronized in accordance with special circumstances. For example, a process might be defined to respond to the **DownTransition** of a mouse button by displaying a menu, and to respond to the **UpTransition** by removing the menu. Interest in the **UpTransition** must be expressed before the **UpTransition** event is automatically distributed by release of the button; however, since the button may be released immediately, distribution of the event must be explicitly delayed until interest has been expressed.

NeWS provides a primitive named **blockinputqueue**, which prevents events from being removed from the global event queue. The syntax is as follows:

num **blockinputqueue** –
The *num* argument specifies the amount of time (in units of 2^{16} ms) during which blocking continues. When the operator is executed, no event is removed from the global event queue until one of the following has occurred:

- The time specified by the *num* argument has elapsed.
- The **unblockinputqueue** operator is executed.

The syntax of the **unblockinputqueue** operator is as follows:

– unblockinputqueue –
This operator releases the event queue lock previously set by **blockinputqueue**. If more than one event queue lock was set, additional calls to **unblockinputqueue** may be required; when all locks are released, the distribution mechanism resumes.

3.16. Event Logging

As a development aid, NeWS provides the **seteventlogger** primitive, which allows you to designate a process as an *event-logger*:

process **seteventlogger** –
The *process* argument must be a process that has expressed some interest and has entered an **awaitevent** loop. The expressed interest, which must not match any distributed event, is required to prevent **awaitevent** from returning a syntax error. The specified process becomes the *event-logger*. A copy of each event either removed from the global event queue or redistributed with **redistributeevent** will be given to this process before it is given to any other (note that the existence of the event-logger does not affect the normal running of the distribution mechanism). When the **awaitevent** loop retrieves the event-copies from the event-logger's local event queue, the event-logger can proceed in whatever way is appropriate. For example, it might print certain key values in a window or to a file.

To turn off a designated event-logger, specify **null** as the argument to **seteventlogger**.

The file `eventlog.ps`, which is described in Chapter 10, *Extensibility through POSTSCRIPT Language Files*. provides a formatted display of events that can be used in the context of the **seteventlogger** operator.

The current event-logger is returned by the **geteventlogger** operator:

– **geteventlogger** process
The operator returns the process that is the current event logger or **null** if there is none.

4

Classes

4 Classes

NeWS provides an object-oriented programming scheme based on classes. The code that implements the basic class mechanism is located in the `class.ps` file (see Chapter 10, *Extensibility through POSTSCRIPT Language Files*, for information about the POSTSCRIPT language files). Client applications will find classes especially useful for creating user interface components such as windows, menus, and scrollbars.

The NeWS class system is extremely flexible. You can define your own classes, so you can build whatever user interface components you desire. You can also use the predefined classes that are supplied with the NeWS toolkit. The classes in the NeWS toolkit provide the building blocks of a user interface; they do not impose any particular style of user interface.

This chapter provides an introduction to the NeWS class system. You should read this chapter if you want to create your own classes or if you are going to use the NeWS toolkit. The toolkit classes are built with the basic class mechanisms described here.

This chapter explains how to use the NeWS class operators and methods. Alphabetical lists of the operators and methods are provided at the end of the chapter.

Special notation is used to help you distinguish between operators and methods. Names of methods are preceded by a slash (for example, **/new**). Names of operators are written without a slash (for example, **send**). Optional arguments to operators and methods are listed in angle brackets (for example, <args>).

4.1. Basic Terms and Concepts

This section explains some basic terms and concepts that are used throughout this chapter. Some of the terms are common object-oriented programming terms; others are specific to the NeWS class system.

Classes and Instances

In the context of classes, an *object* consists of data and the procedures needed to operate on that data. NeWS represents these objects as POSTSCRIPT language dictionaries. An object's dictionary contains the object's data (represented as variables) and the object's procedures (represented as POSTSCRIPT language procedures).

A *class* is a template for a set of similar objects; the objects described by the class are known as *instances* of the class. An instance of a class *inherits* the characteristics of its class but can selectively alter some of these characteristics.

Classes and instances of classes are all objects; they are all represented by POSTSCRIPT language dictionaries that store the object's variables and procedures.

A class is like an architect's plan for a house; it is a blueprint that specifies the fundamental characteristics of a specific type of object. An instance of the class is like the house itself; it is a particular object that is based on the blueprint.

When you create a class, you must specify its *instance variables*, *class variables*, and *methods*. All of these items are stored in the class' dictionary. Each variable is stored with its variable name as a dictionary key and its variable value as the dictionary key's value. Each method is stored with its name as a dictionary key and its procedure as the dictionary key's value.

A class' instance variables are variable data contained in each instance of the class. Each instance receives its own copy of its class' instance variables, and each instance is free to change the values associated with its copy of the instance variables. The instance variables are stored in an instance dictionary in the same way that they are stored in a class dictionary: each variable name-value pair is stored as a key-value pair in the instance dictionary.

Class variables are variable data shared by all the instances of a class. A class' class variables are stored in its class dictionary, but the instances of the class do not receive a copy of the class variables. If you change the value of a class variable, that change affects all the instances of the class.

A class' methods are procedures that you use to operate on the class' instances. You send a *message* to an object to invoke the method associated with that message; the message identifies the name of the method that you want to invoke. Class methods are stored only in class dictionaries, not in instance dictionaries.

To continue the house analogy, assume that a whole subdivision of houses is built with the same blueprint. The houses have the same floor plan and the same style, but each house is slightly different. For example, the paint and carpet colors vary from house to house. Instances of a class are like the houses in the subdivision; the instances have certain basic characteristics in common, and they perform the same functions, but each instance is slightly different.

In this analogy, the physical aspects that vary from house to house correspond to the instance variables. The physical aspects that are specified in the blueprint, and thus do not vary from house to house, correspond to the class variables. The blueprint also specifes certain functions that all the houses must perform. For example, each house must provide a working electrical system, plumbing system, and heating system. These functions specified in the blueprint correspond to the class methods. The "messages" that someone must send to invoke these functions of a house are flipping on an a light switch, turning on a faucet, and turning up the thermostat.

Inheritance and the Class Tree

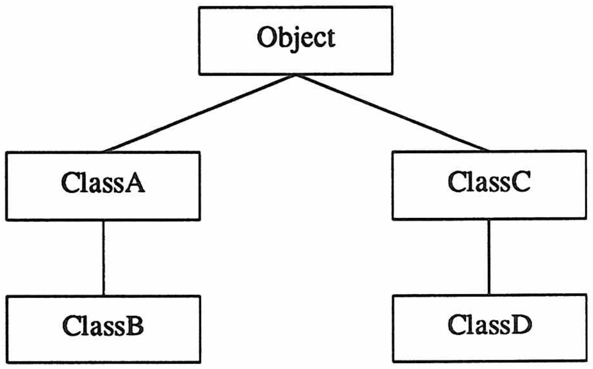
The classes in the NeWS class system belong to a class tree. The class tree is a hierarchy that is similar to, but completely separate from, the canvas tree. The root of the class tree is class **Object**. The server provides the implementation of class **Object** (in the `class.ps` file), and the other classes in the tree are defined by the client or by a toolkit.

Superclasses and Subclasses

Except for class **Object**, each class has at least one class that is above it on its branch of the class tree; these classes that are above a class are called the class' *superclasses*. A class can also have *subclasses*, which are located on branches that emanate from beneath the class. Thus, a class' superclasses are closer to the root of the class tree, and a class' subclasses are farther from the root.

The illustration below shows the structure of a simple class tree with class **Object** at the root of the tree. This tree has just two short branches.

Figure 4-1 *A simple class tree*



In this example, ClassA and ClassC are subclasses of class **Object**. **Object** is the superclass of ClassA and ClassC. ClassB is a subclass of ClassA, and ClassD is a subclass of ClassC. ClassB and ClassD each have two superclasses: ClassB's superclasses are ClassA and class **Object**, and ClassD's superclasses are ClassC and class **Object**.

The Immediate Superclass

The superclass that is immediately above a class on its branch of the class tree is called the class' *immediate superclass*. ClassB's immediate superclass is ClassA, and ClassD's immediate superclass is ClassC. ClassA and ClassC both have class **Object** as an immediate superclass.

Inheritance

A class inherits the variables and methods of all its superclasses. For example, ClassB inherits all the variables and methods of ClassA and class **Object**. Note that class **Object**'s methods are available to all classes in the tree since **Object** is the root of the tree.

A class can override any of the variables and methods that it inherits. For example, ClassB can redefine a variable or method that is defined in ClassA. When a subclass overrides a method of one of its superclasses, the subclass can simply add to the method definition given by the superclass, or it can completely redefine the method. A class can also define new variables and methods.

An instance inherits the variables and methods of its class and its class' superclasses. For example, an instance of ClassB inherits the variables and methods of ClassB, ClassA, and **Object**.

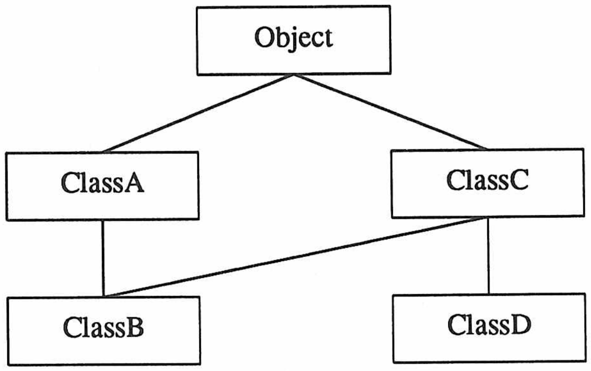
An instance can override anything that it inherits, although it usually should not override a class variable or method. An instance often changes the values associated with the instance variables that it inherits. In unusual cases, an instance can even define new variables and methods.

Single Inheritance and Multiple Inheritance

Two kinds of inheritance can occur in the class tree: *single inheritance* and *multiple inheritance*. The term single inheritance refers to the case in which a class has only one immediate superclass. The term multiple inheritance refers to the case in which a class has more than one immediate superclass.

The class tree shown in the previous figure contains only single inheritance because all of the classes have only one immediate superclass. An example of multiple inheritance would be if ClassB inherited not only from ClassA, but also from ClassC. In this case, another line would need to be drawn on the tree diagram to connect ClassB to ClassC. This situation is illustrated below.

Figure 4-2 *A class tree with multiple inheritance*



In this example of multiple inheritance, ClassB has three superclasses: ClassA, ClassC, and **Object**. ClassB has two immediate superclasses: ClassA and ClassC.

ClassB inherits from all three of its superclasses. But a question arises: should ClassA override ClassC or vice versa? This issue is discussed in detail in Section 4.12, *Multiple Inheritance*.

The Inheritance Array

When you create a class, you must specify where the class belongs in the class tree; you do this by specifying the new class' immediate superclass(es). In the single inheritance case, you just need to specify the one class that is immediately above the new class. In the multiple inheritance case, you need to specify all the class' immediate superclasses.

Based on this immediate superclass information for the new class, NeWS creates a special array called the class' *inheritance array*. The inheritance array lists all the class' superclasses in the order that they override each other. Each class in the array overrides the classes listed after it in the array.

In the single inheritance case, a class' inheritance array contains all the class' superclasses listed in leaf-to-root order. For example, the inheritance array of ClassD is

[ClassC Object]

and the inheritance array of ClassA is

[Object}

In the multiple inheritance case, a class' inheritance array still contains all the class' superclasses, but a unique order no longer exists. A valid inheritance array consists of any arrangement of the superclasses that maintains the leaf-to-root order of classes on the same branch. For example, ClassB in the above figure has the following two possible inheritance arrays:

[ClassA ClassC Object]

[ClassC ClassA Object]

You can choose either one of these arrays for ClassB. Section 4.12, *Multiple Inheritance*, explains the details of inheritance arrays for the multiple inheritance case.

Each instance also has an inheritance array. An instance's inheritance array is the same as the inheritance array of its class except that the class is added to the list. Thus, an instance's inheritance array contains its class and all of its class' superclasses. For example, the inheritance array of an instance of ClassD is

[ClassD ClassA Object]

and the inheritance array of an instance of ClassA is

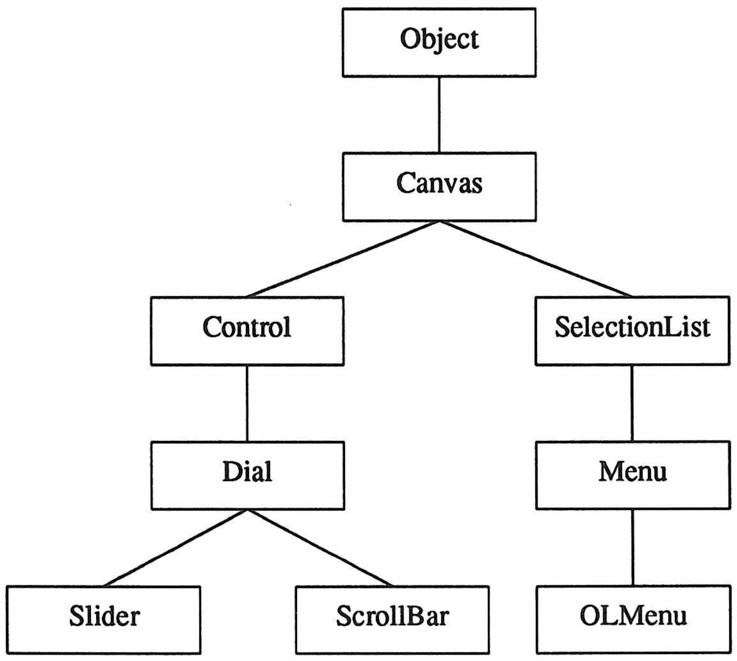
[ClassA Object]

An instance has a copy of all the instance variables of the classes in its inheritance array, and an instance can invoke any of the methods of the classes in its inheritance array.

A Single Inheritance Example

This section describes a single inheritance example in which every class has only one superclass. The following figure illustrates the class tree for this example.

Figure 4-3 *A single inheritance example*



In this example, class **Object** has one immediate subclass named class Canvas. Class Canvas and its subclasses implement different kinds of canvases, such as menus and scrollbars. Note that the class tree should not be confused with the canvas tree. Instances of class Canvas (and of its subclasses) represent NeWS canvas objects that exist in the canvas tree. But the instances inherit their variables and methods from class Canvas in the class tree.

In this example, class Control handles the basic user interaction operations needed by control objects such as dials. Control objects are canvases that have a current value and a callback procedure; the callback procedure is executed when the user interacts with the object to change its current value.

Class Dial is a subclass of Control that provides the basic operations needed to build various types of dials. A dial lets the user choose a numeric value between a minimum and maximum. Sliders and scrollbars are types of dials. Scrollbars are commonly used to scroll through a text file. Class Slider implements generic sliders, and class ScrollBar implements scrollbars.

Class SelectionList manages a list of items, along with any sublists the items have; this class provides the basic operations needed by menus. Class Menu implements a basic menu, using the operations defined in SelectionList. Class OLMenu is used to create menus with a special user interface.

You can arrange your class tree (your subclasses) to maximize modularity and to take advantage of the shared aspects of objects. You can implement different variations of an object as subclasses of one class. For example, you might have several different user interface options for menus; each user interface option could be a subclass of class Menu. Class Menu would contain code that is common to all menus, thus avoiding repetition of the same code in each type of menu object.

Since this example is a single inheritance case, every class has just one immediate superclass. For example, class Dial's immediate superclass is Control, and class Control's immediate superclass is Canvas.

In the single inheritance case, the inheritance array for any class consists of the class' superclasses, listed in leaf-to-root order. For example, class ScrollBar's inheritance array is

[Dial Control Canvas Object]

and class Menu's inheritance array is

[SelectionList Canvas Object]

Assume that you have an instance of class ScrollBar named MyScrollBar and an instance of class OLMenu named MyOLMenu. The inheritance array of an instance is the same as the inheritance array of the instance's class, except that the instance's class is added to the array. For example, the inheritance array for MyScrollBar is

[ScrollBar Dial Control Canvas Object]

and the inheritance array for MyOLMenu is

[OLMenu Menu SelectionList Canvas Object]

Summary of Terms

The following table summarizes the class terminology introduced in the previous sections.

Table 4-1 *Summary of Terms*

object	a class or an instance; each class and instance object consists of variables and procedures stored in a POSTSCRIPT language dictionary
class	a template for a set of similar objects known as instances
instance	one of the objects described by a class; an instance inherits its variables and procedures from its class
instance variables	variables that are given to each instance of a class
class variables	variables that are shared by all instances of a class
methods	procedures that a class uses to operate on its instances
message	a method name that is sent to an object to invoke the associated method
Object	the class that is the root of the class tree
superclasses	a class' superclasses are located on the branch(es) that emanate root-ward from the class (in the single inheritance case only one such branch exists and it connects the class to the root); a class inherits from all its superclasses
subclasses	a class' subclasses are located on the branches that emanate leaf-ward from the class
single inheritance	when a class' superclasses all occupy the same branch of the tree
multiple inheritance	when a class' superclasses do not all occupy the same branch of the tree
immediate superclass	in the single inheritance case, a class' immediate superclass is directly above the class on the branch that connects the class to the root; in the multiple inheritance case, a class has more than one branch that emanates root-ward from the class and each such branch has an immediate superclass that is directly above the class
inheritance array	each object has an inheritance array that contains the classes from which the object inherits, listed in the order that the classes override each other

4.2. Creating a New Class

To create a new class, you use the **classbegin** and **classend** operators in sequence.

The Class Definition

The basic structure of a class definition is given below (you define each class variable and method with the **def** operator).

```
/classname [superclasses] [instancevars]
classbegin
        class variable definitions
        class method definitions
classend def
```

The operators that are used in class definitions are described below.

classbegin

classname superclasses instancevars **classbegin** —

Creates an empty class dictionary for the new class and puts it on the dictionary stack.

classbegin takes three arguments: the classname, the immediate superclass or an array of superclasses, and the instance variables. You specify the superclass(es) as one immediate superclass (the single inheritance case) or as an array of superclasses (the multiple inheritance case). See Section 4.12, *Multiple Inheritance*, for an explanation of how to specify an array of superclasses. You can specify the instance variables as an array of names or as a dictionary of key-value pairs. If you use an array of names, the variables are initialized to null; if you use a dictionary, the variables are initialized to the values specified in the dictionary.

After calling **classbegin**, you use the **def** operator to fill the class dictionary with the class' variables and methods. Then you call **classend** to complete the creation of the new class.

classend

— **classend** classname newclass

Completes the class dictionary that was left on the dictionary stack by **classbegin**. The **classend** operator constructs the inheritance array based on the superclass(es) that you passed to **classbegin** (see Section 4.12, *Multiple Inheritance*, for a discussion of the inheritance array in the multiple inheritance case). **classend** also compiles the class' methods (see Section 4.5, *Method Compilation*) and executes any procedures in **UserProfile** that have the same name as the class (see Section 4.8, *Overriding Class Variables With UserProfile*). **classend** returns the name of the new class (the name that you passed to **classbegin**) and the new class dictionary.

redef

name object **redef** —

In a class definition, the **redef** operator redefines an instance variable that is already defined in one of the class' superclasses. If you use the **def** operator to redefine an instance variable in a dictionary passed to **classbegin**, you will be warned that you are redefining an existing instance variable. If you want to avoid the warning, you must use the **redef** operator instead of the **def** operator.

Initializing a New Class

If a class requires some processing before the definition of the class is complete, the convention is to put the initialization code in a **/classinit** method for the class. For example, class **Object**'s **/classinit** method starts a process that listens for obsolescence events; class **Object** then handles obsolete classes and instances as explained in Section 4.11, *Obsolete Objects in the Class System.*

4.3. Sending Messages With the send Operator

This section explains how to use the **send** operator to invoke class methods. The section discusses both forms of the **send** operator and gives an example of a simple **send** and a nested **send**.

The Usual Form of send

<args> name object **send** <results>

Sends a message to an object to invoke the method associated with the message. The *name* argument is the name of the method that is invoked by the message, and the *object* argument is the receiver of the message. The *object* argument is often an instance, but it can also be a class. Any arguments required by the method must be specified; any results of the method are returned.

Before **send** invokes the *name* method, it places the classes in *object*'s inheritance array on the dictionary stack and places *object* on top of the dictionary stack. When the *name* method is invoked, the server searches the stack from top to bottom to find the method; the server finds the first occurrence of the method in the inheritance array that is on the stack. This mechanism ensures that classes override each other in the proper order. After the *name* method executes, the **send** operator restores the dictionary stack to the state it was in before the **send**.

Thus, the **send** operator takes advantage of the stack-based nature of the POSTSCRIPT language to implement inheritance. An object can access the class variables and methods of the classes in its inheritance array because the object's inheritance array is placed on the dictionary stack when a message is sent to that object. This arrangement allows a class dictionary to store only its own class variables and methods, not the class variables and methods of its superclasses. Likewise, an instance only needs to store its instance variables.

The group of objects that is put on the dictionary stack during a **send** is known as the **send** *context*. The **send** context includes the message receiver and the classes in its inheritance array.

The **send** process is explained in detail below.

The Steps Involved in a send

When *name* is sent to *object*, the following steps are taken:

1. Any existing **send** context is temporarily removed from the dictionary stack. (In a nested **send**, the first **send**'s context is on the dictionary stack when the second **send** is called.) If a local dictionary happens to be on top of the dictionary stack (because **send** is called inside the local dictionary), then **send** temporarily removes the local dictionary from the stack. The example in the subsection *A Nested send* illustrates how **send** handles local dictionaries.

Note that you might have problems if one of your methods puts a local dictionary on the stack and never removes it from the stack. See Section 4.5, *Method Compilation*, if you plan to use such a method; you may need to take special precautions to ensure that the local dictionary is handled properly.

2. The **send** operator establishes *object*'s context by putting *object* and all of the classes in *object*'s inheritance array onto the dictionary stack. The inheritance array is placed on the stack with the root-most classes toward the bottom of the stack and the leaf-most classes toward the top; *object* itself is placed on the top of the stack.
3. The server searches the dictionary stack from top to bottom for the *name* method. Because *object* and the classes in its inheritance array were placed on the dictionary stack, the server finds the first occurrence of the method in *object*'s context. If the chain of classes is searched all the way back to the root without finding the specified method, an error is returned.
4. When the *name* method is found, it is executed. The arguments required by the method are taken from the operand stack, and any results of the method are put on the operand stack.
5. The initial context is then restored; the dictionary stack is restored to the state it was in before the **send** was made. If a local dictionary was removed from the top of the stack in step 1, the local dictionary is restored to its original position at the top of the stack.

The example in the following section illustrates these five steps.

Using send to Invoke a Method

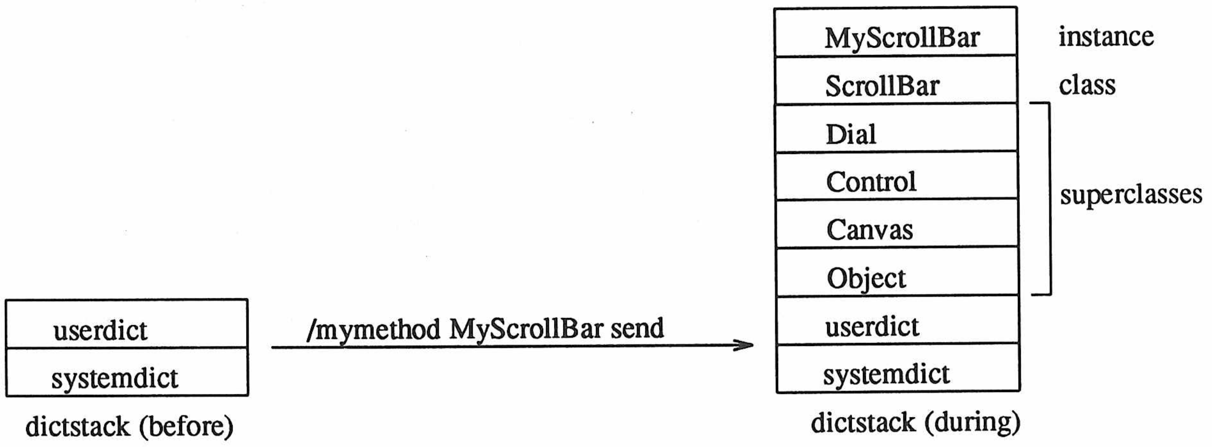
This example uses the class hierarchy given in Section 4.1, *Basic Terms and Concepts*. Assume that **send** is invoked as follows:

```
arg1 arg2 /mymethod MyScrollBar send
```

Also assume that before this **send**, the dictionary stack contains the **systemdict** on the bottom and the **userdict** on the top. When this **send** is executed, the following steps are taken:

1. No existing **send** context is on the stack when this **send** is called, so nothing is removed from the stack.
2. The instance MyScrollBar and the classes in its inheritance array are pushed on the dictionary stack, as shown in the following figure:

Figure 4-4 *Dictionary stack before and during a* send *to* MyScrollBar



3. The server locates /mymethod in one of the classes on the stack.
4. The server executes /mymethod. As /mymethod executes, it consumes arg2 and arg1 from the operand stack. If /mymethod returns any results, they are placed on the operand stack.
5) The **send** operator restores the dictionary stack to its previous state with the **systemdict** on the bottom and the **userdict** on the top.

A Nested send

This section expands on the previous example to illustrate a nested **send** (one **send** within another). The example also shows what happens when **send** is used in a local dictionary.

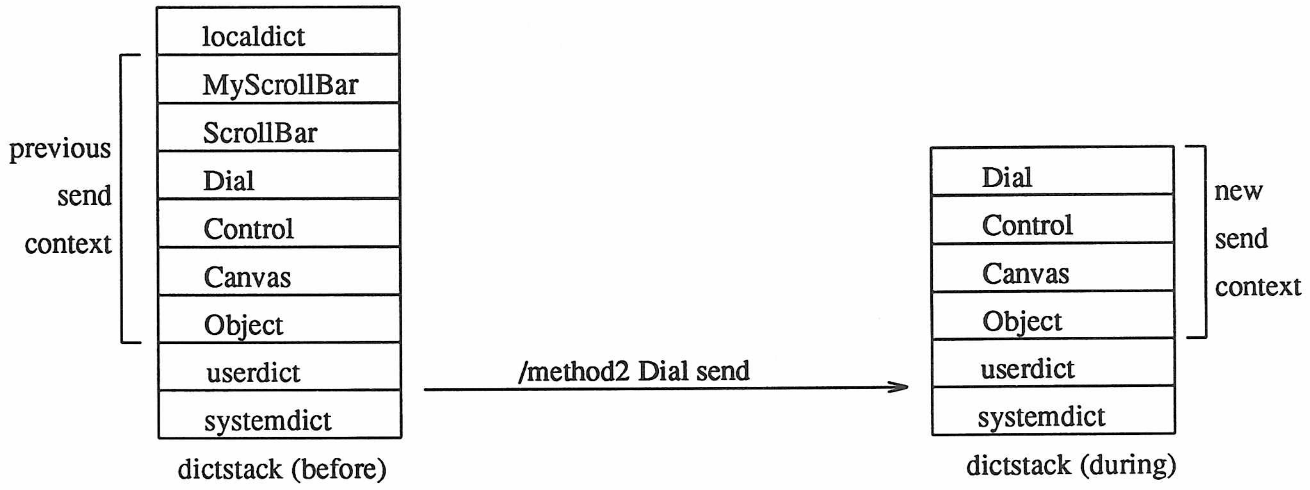
Assume that /mymethod is sent to MyScrollBar as before. The classes in MyScrollBar's inheritance array are put on the dictionary stack. Suppose that /mymethod is located in ScrollBar and that /mymethod is defined as follows:

```
/mymethod {
        10 dict begin
                .
                .
                /method2 Dial send
                .
                .
        end
        .
        .
} def
```

When /mymethod is found and executed, it puts a local dictionary on the dictionary stack. When the **send** to Dial is encountered in /mymethod, the following steps are taken:

1. This inner **send** removes the local dictionary and the existing **send** context (MyScrollBar and its inheritance array) from the dictionary stack.
2. The **send** to Dial then puts Dial and its inheritance array on the stack, as shown in the following figure:

Figure 4-5 *Dictionary stack before and during a nested* **send**



3. The server locates /method2 in one of the classes on the stack.
4. The server executes /method2.
5. The inner **send** takes its **send** context (Dial and its inheritance array) off the stack and puts the previous **send** context (MyScrollBar and its inheritance array) back on the stack. The local dictionary is placed back on top of the stack.

After the inner **send** is complete, /mymethod finishes executing. When **/mymethod** finishes, MyScrollBar and the classes in its inheritance array are removed from the stack to complete the outer **send**.

This example is only meant to illustrate the manipulation of the dictionary stack during a nested **send**. In /mymethod, you would not actually send a message directly to ScrollBar's superclass. Instead, you would use the **super** psuedo-variable to represent the message receiver; **super** is discussed in Section 4.4, *The Psuedo-Variables self and super.*

Using send to Create a New Instance

Class **Object** provides several methods for creating new instances of a class. The /**new** method is briefly introduced here; the creation of new instances is discussed in detail in Section 4.6, *Creating a New Instance.*

The following example creates a new instance of MyClass by sending the /**new** message to MyClass.

```
/new MyClass send
```

In this case, **send** puts MyClass and its inheritance array on the dictionary stack. The server locates the /**new** method and executes it, leaving the new instance on the operand stack. Then **send** removes MyClass and its inheritance array from the dictionary stack.

Another Form of send

<args> proc object **send** <results>

Executes *proc* in the context of *object*, exactly as if *proc* had been predefined as a method and given a name that was passed as an argument to **send**. Any arguments needed by the procedure are taken from the operand stack; any results of the procedure are returned to the operand stack. The syntax for this form of **send** is shown below.

```
{procedure} object send
```

The /**doit** method must sometimes be used in conjunction with this form of send. For details, see Section 4.5, *Method Compilation*.

This form of **send** bypasses the established class interface and should rarely be used. One valid use of this form of **send** is a *batch* **send**; see the /**doit** method in Section 4.5, *Method Compilation*, for an example of a batch **send**.

Using send to Change the Value of an Instance Variable

After you create a class and some instances of the class, you will probably want to change the values of some of the instance variables. Although you can change the value of an instance's variable in several ways, only one way is proper.

The appropriate way to change the value of an instance variable is to include in the class definition a method that changes the value. Then you can send that message to any instance of the class to change the value of its copy of that instance variable. This is just a specific case of using **send** to invoke a class method.

You can also change the value of an instance variable by passing a new value in a procedure argument to **send** (see the subsection *Another Form of send*, above) or by putting the value directly in the instance dictionary. Both these methods are discouraged because they ignore the established class interface. A class method that changes the value of an instance variable might also take a special action when the value is changed. For example, suppose class Dial has a /setvalue method that not only sets the Dial's internal value, but also redraws the dial on the screen to reflect the new value. For this reason, you should always use an established class interface to change the value of instance variables.

Using send to Change the Value of a Class Variable

You change the value of a class variable the same way you change the value of an instance variable: define a class method that changes the value of the variable, and then invoke the method. Note that you should use **store** instead of **def** in methods that define the value of class variables. If you used **def**, you might accidently add the class variable to an instance dictionary that happens to be on top of the stack. (You can intentionally add a class variable to an instance dictionary; this action is known as *promoting* the instance variable. For details, see

Section 4.9, *Promoting Class Variables to Instance Variables.*

4.4. The Psuedo-Variables self and super

When **send** is used outside a method, an object is given as an argument to **send**, and the search for the method begins with that object. The object argument to **send** can be an instance or a class.

When **send** is used inside a method, two special symbols named **self** and **super** can be used as the object argument to the **send** operator. These symbols, known as *psuedo-variables*, add flexibility and generality to class methods because they take different values depending on the situation.

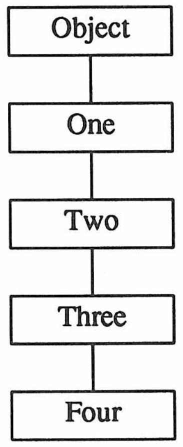
This section uses a simple example to illustrate **self** and **super** (this example is adapted from an example in Adele Goldberg's *SmallTalk — The Interactive Programming Environment*, Addison Wesley, 1984, pp 62-66).

Four classes are defined as follows:

```
/One Object [ ] classbegin
        /test {1} def
        /result1 {/test self send} def
classend def

/Two One [ ] classbegin
        /test {2} def
classend def

/Three Two [ ] classbegin
        /result2 {/result1 self send} def
        /result3 {/test super send} def
classend def

/Four Three [ ] classbegin
        /test {4} def
classend def
```

Class **Object** has a subclass named One, class One a subclass named Two, class Two a subclass named Three, and class Three a subclass named Four. The following diagram illustrates this simple class tree:

Figure 4-6 *Class tree for* **self** *and* **super** *example*



These classes do not define any instance or class variables, but they do define some methods. The method definitions are summarized below.

- Class One defines a method named /test that puts the number 1 on the operand stack. Class One also defines a method named /result1 that sends the /test message to **self.**
- Class Two defines a method named /test that puts the number 2 on the operand stack. Class Two's /test method overrides class One's /test method.
- Class Three defines a method named /result2 that sends /result1 to **self.** Class Three also defines a method named /result3 that sends /test to **super.**
- Class Four defines a method named /test that puts the number 4 on the operand stack. Class Four's /test method overrides the /test methods in classes One and Two.

An instance of each class is created as shown below. Inst1 is an instance of class One, Inst2 an instance of class Two, Inst3 an instance of class Three, and Inst4 an instance of class Four.

```
/Inst1 /new One send def
/Inst2 /new Two send def
/Inst3 /new Three send def
/Inst4 /new Four send def
```

The `psh` command can be used to begin an interactive session with NeWS (see the manual page for `psh` in the *X11/NeWS Server Guide*). The above class and instance definitions can be defined during such an interactive session. Then the class methods can be executed by sending messages to the instances. The next two sections use this approach to illustrate **sends** to **self** and **super** for these class and instance definitions. For each example **send,** the code that is typed to `psh` is shown on the first line (in sans serif font) and the resulting number that the server prints to the screen is shown on the second line (in `listing` font).

The self Psuedo-Variable

When a message is sent to **self**, the search for the method begins with the object that received the original message that caused the current method to be invoked. Thus, **self** represents the object that is on top of the dictionary stack at the time that the **self send** is encountered. The following examples clarify the use of **self**.

First, the /result1 message is sent to Inst1 as follows:

```
/result1 Inst1 send =
1
```

When /result1 is sent to Inst1, the following actions are taken:

1. The **send** operator puts Inst1 and the classes in its inheritance array on the dictionary stack.
2. The /result1 method is found in class One and is executed. The /result1 method sends /test to self, which in this case is Inst1. (The instance Inst1 is the object that received the message, /result1, that caused the /test method to be invoked.)
3. Because this is a nested **send**, the old **send** context is temporarily removed from the dictionary stack, and the new **send** context is put on the dictionary stack. In this case, the old and new **send** contexts are identical since the **sends** were made to the same object; the first message (/result1) was sent to Inst1, and the second message (/test) was sent to **self**, which resolved to Inst1. When the new context is put on the stack, the stack still contains Inst1 and its inheritance array.
4. The search for the /test method begins with **self**, which is Inst1. The /test method is found in class One. When executed, /test puts the number 1 on the operand stack.
5. The two nested **send** contexts are then cleared from the dictionary stack. First the new **send** context is removed and replaced with the old context; then the old **send** context is removed to complete the outer **send**.
6. After the **sends** are completed, the number 1 is printed to the screen with the = operator.

Note that the context swapping in this nested **send** is inefficient. The same context is swapped on and off the stack several times. The NeWS class mechanism usually avoids doing these extra context swaps that occur when **self** is used with **send**; the **classend** operator *compiles* a class' methods to replace most occurrences of /method self send with a more efficient form (see Section 4.5, *Method Compilation*). Because **self** is implemented as an operator that returns an object, the construct /method self send can be executed even if it is not compiled; the compilation is done merely as an optimization.

Next, the /result1 message is sent to Inst2 as follows:

```
/result1 Inst2 send =
2
```

When /result1 is sent to Inst2, the following actions are taken:

1. The **send** operator puts Inst2 and the classes in its inheritance array on the dictionary stack.
2. The /result1 method is found in class One. The /result1 method sends /test to **self**, which in this case is Inst2. Thus the search for the /test method begins with Inst2, in the same context.
3. The /test method is found in class Two. When executed, /test puts the number 2 on the operand stack.
4. The dictionary stack is restored to its initial state with the **systemdict** on the bottom and the **userdict** on the top.
5. The number 2 is printed to the screen with the = operator.

Below are four more example **sends**.

```
/test Inst3 send =
2
/result1 Inst4 send =
4
/result2 Inst3 send =
2
/result2 Inst4 send =
4
```

The super Psuedo-Variable

The **super** psuedo-variable provides a way to invoke a method that would otherwise be overridden. If **super** is used in a method as the object argument to **send**, the search for the method associated with **send**'s message begins with the class that is immediately below the method's class on the dictionary stack (the next superclass in the current **send** context). In other words, **super** represents the class that follows the method's class in the inheritance array that is currently on the dictionary stack.

The next two examples use the same class and instance definitions as the previous section, but this time they illustrate the **super** psuedo-variable.

First, the /result3 message is sent to Inst3 as follows:

```
/result3 Inst3 send =
2
```

When the /result3 message is sent to Inst3, the following actions are taken:

1. The **send** operator puts Inst3 and the classes in its inheritance array on the dictionary stack. The dictionary stack then contains, from bottom to top, the

systemdict, the userdict, class Object, class One, class Two, class Three, and Inst3.

2. The /result3 method is found in class Three. The /result3 method sends /test to super, which in this case is class Two. Note that super is the class that follows /result3's class in the current send context, not the class that follows Inst3.

3. Like any nested send, the send to super involves an old send context and a new send context. In this case, the old send context is Inst3 and its inheritance array. The new send context is super, or class Two, and its inheritance array. These two contexts are identical except that the new context begins with class Two instead of Inst3; the chain of superclasses is the same, but the new context just omits class Three and Inst3. Therefore, the contexts do not need to be swapped, as long as the search for the method begins with super rather than with the object on top of the stack.

   The search for the /test method begins with super, which is class Two.

4. The /test method is found in class Two. When /test is executed, it puts the number 2 on the operand stack.

5. The dictionary stack is restored to its initial state with the systemdict on the bottom and the userdict on the top.

6. The number 2 is then printed to the screen with the = operator.

Unlike **self**, **super** is not implemented as an operator that returns an object. When the **classend** operator compiles a class' methods, each occurrence of /method super send is replaced with an operator that resolves **super** and then finds and executes the method in the current context. Thus **super** cannot be used without **send**, and it cannot be used unless the method in which it occurs is compiled. As a consequence of this implementation, the context swapping is always avoided for **sends** to **super** (see Section 4.5, *Method Compilation*).

Now the /result3 message is sent to Inst4 as follows:

```
/result3 Inst4 send =
2
```

When the /result3 message is sent to Inst4, the following actions are taken:

1. The **send** operator puts Inst4 and the classes in its inheritance array on the dictionary stack. The dictionary stack then contains, from bottom to top, the **systemdict**, the **userdict**, class **Object**, class One, class Two, class Three, class Four, and Inst4.

2. The /result3 method is found in class Three. The /result3 method sends /test to **super**, which is class Two. The search for /test begins with class Two, in the same context.

3. The /test method is found in class Two. The /test method is executed, putting the number 2 on the operand stack.

4. The dictionary stack is restored to its initial state with the **systemdict** on the bottom and the **userdict** on the top.

5. The number 2 is printed to the screen with the = operator.

Using super to Send a Message Up the Superclass Chain

The **super** pseudo-variable is often used recursively to send a message up the superclass chain. If a method sends a message to **super**, the method in **super** can send the same message to its **super**, and the **sends** to **super** can continue until the root of the class tree is reached.

This construction allows a subclass to add to a method of one of its superclasses without repeating the entire code of the method. The subclass' method can first send the method to **super** to execute its superclass' operations for that method; then the subclass' method can add its own sequence of operations to its definition of the method. If all the classes on the branch define the method in this way, the message will pass all the way up the class chain to the root.

Below is the basic structure used in a method to send a message up the superclass chain:

```
/mymethod {
        /mymethod super send        % Do what super does.
        ...                         % Do what this class wants to do.
} def
```

Restrictions on the Use of self and super

In addition to being used as an argument to **send**, **self** can be used anywhere in a class definition to refer to the object that **self** represents. This usage is possible because **self** is implemented as an operator that puts an object on the stack.

Unlike **self**, **super** can only be used as an argument to **send**. The **super** psuedo-variable is not implemented as an operator that returns an object; for details on how **super** is implemented, see Section 4.5, *Method Compilation*. The **super** psuedo-variable has one other restriction on its use: **super** cannot be used anywhere in a procedure passed to **send** unless the **/doit** method is used (see **/doit** in Section 4.5, *Method Compilation*).

4.5. Method Compilation

This section is optional reading; it will be helpful to advanced users, but most users will not need the detailed information described here. The one possible exception is the description of batch sends and the **/doit** method; a batch send is a fairly useful concept.

As explained in the examples of **self** and **super** above, **sends** to **self** and **super** can be optimized by leaving the existing context alone. The **classend** operator compiles a class' methods to substitute a more efficient form for most occurrences of **self send** and all occurrences of **super send**. When the methods are invoked later, the context swapping is avoided. Note that **super send** must be compiled, but **self send** is compiled merely as an optimization.

Compiling self send

The method compiler replaces most occurrences of /method self send with method. The search for /method then starts at the top of the existing dictionary stack. The method compiler does not replace /method self send when it occurs in a local dictionary, as explained below.

Compiling super send

The method compiler replaces occurrences of /method super send with an operator that resolves **super** and then finds and executes /method in the current context. The search for /method begins with the object that **super** represents. If /method super send occurs in a local dictionary, the method compiler replaces it with a slightly less efficient form as explained below.

Local Dictionaries

When a **send** is executed, any current **send** context is cleared from the dictionary stack, and the context for the message receiver is established on the dictionary stack. The **send** operator puts the message receiver on top of the dictionary stack. During execution of the method invoked by the **send**, the topmost dictionary is almost always the message receiver. However in certain cases, a method may use a *local dictionary* during its execution. A local dictionary is a dictionary that the method places on the dictionary stack while the method is executing. If a local dictionary is on the stack when a nested **send** is invoked, the local dictionary is removed from the stack before the nested method is invoked (see Section 4.3, *Sending Messages With the send Operator*).

During most **sends**, an instance dictionary is on top of the dictionary stack. Most methods assume that the top dictionary on the dictionary stack is an instance dictionary. That is, most methods assume that they can store into instance variables using the following construct: /variable value def. If a local dictionary were on the stack above the instance dictionary, this construct would make a new value in the local dictionary instead of replacing the instance variable in the instance dictionary; that is why **send** removes local dictionaries before executing a nested method.

In the following example, /method1 pushes a local dictionary mydict onto the dictionary stack and then invokes /method2.

```
/method1 {
        mydict begin
                /method2 self send
        end
} def

/method2 {
        /variable 5 def
} def
```

During the execution of /method2, mydict is not present on the stack because the **send** temporarily removes it, along with the previous **send** context. Thus when /method1 is sent to an instance, variable is stored in the instance dictionary.

The method compiler usually replaces /method self send with method. This substitution works when the topmost dictionary is the message receiver. However, this optimization fails in the presence of local dictionaries. Returning to the example, the following code illustrates the problem that would occur if the method compiler optimized /method2 self send:

```
/method1 {
        mydict begin
                method2
        end
} def

/method2 {
        /variable 5 def
} def
```

In this case, mydict would still be on the dictionary stack when /method2 is invoked. As a result, variable would be stored into mydict instead of being stored as an instance variable.

To avoid this problem, the method compiler does not replace **self send** when it occurs within a local dictionary. The method compiler still replaces **super send** when it occurs in a local dictionary, but it uses a slightly less efficient form to ensure that the local dictionary is handled properly.

The method compiler keeps track of local dictionaries in methods by counting **begin/end** and **dictbegin/dictend** pairs. When the method compiler starts to compile a method, the counter is initialized to zero. Each time a **begin** or **dictbegin** is encountered, the count is incremented by one; each time an **end** or **dictend** is encountered, the count is decremented by one. If the count is less than or equal to zero when the method compiler comes across a **self send** or **super send**, the compiler substitutes the most efficient form.

Controlling Method Compilation

The method compiler can be fooled if you have a method that pushes a local dictionary on the stack and does not remove it. You can compensate for this situation with the **SetLocalDicts** compiler directive. You can also use **SetLocalDicts** to force the method compiler to optimize a **self send** or **super send** in a local dictionary (if you want to purposely leave the dictionary on the stack). For details, see the explanation of **SetLocalDicts** below.

Three methods are available to compile a method outside of a class definition. These three methods and the **SetLocalDicts** directive are described below.

/methodcompile

uncompiledproc **/methodcompile** compiledproc

Compiles a procedure to replace occurrences of **self send** and **super send** as discussed above. **/methodcompile** is called by **classend** to compile a class' methods; it can also be used directly to compile a procedure that is passed to it. The following example compiles a procedure in the context of MyClass and returns the new, compiled, executable array:

```
{procedure} /methodcompile MyClass send
```

/installmethod

name proc /**installmethod** —

Creates a new method outside of a class definition. When you send **/installmethod** to an object, it installs the specified procedure as a method of the object and gives the method the specified *name*. **/installmethod** compiles the procedure by calling **/methodcompile**, and then it adds the method to the object's dictionary. The object can be a class or an instance; in the latter case, **/installmethod** creates an "instance method."

In the example below, a new method called /mymethod is installed in MyClass.

```
/mymethod {procedure} /installmethod MyClass send
```

/doit

<args> proc /**doit** <results>

Compiles and executes a procedure. The **/doit** method is used to compile a procedure that is passed to the **send** operator (see *Another Form of send* in Section 4.3, *Sending Messages With the send Operator*, above). You use **/doit** in the following way:

```
{procedure} /doit myinstance send
```

If you use the procedure form of **send** outside of a method, the following rules apply:

- **/doit** is required when the procedure passed to **send** contains a reference to **super**.
- **/doit** is suggested when the procedure passed to **send** contains a reference to **self**. Although the **send** works without **/doit** in the case of **self**, the **send** is more efficient when you compile the procedure.

If you use the procedure form of **send** inside a method definition, you do not need to use **/doit** because any **self sends** and **super sends** are compiled when the method is compiled.

The procedure form of **send** is commonly used with **/doit** to send a group of messages, or a *batch* **send**, to an object. The following example sends four messages to myinstance.

```
{
/method1 self send
/method2 self send
/method3 self send
/method4 self send
} /doit myinstance send
```

The above code is more efficient than sending each message separately to myinstance because only one **send** is actually executed; the **sends** to **self** are avoided by the method compiler. Note that **/doit** could be omitted if the above batch **send** was located inside a method definition.

A batch **send** can omit both the **/doit** method and the **self sends**, as follows:

```
{
method1
method2
method3
method4
} myinstance send
```

However, the above construction is not as clear as the **self send** form and is therefore not recommended.

SetLocalDicts

int **SetLocalDicts** —

Sets the method compiler's local dictionary count to *int*. When the local dictionary count is less than or equal to zero, the method compiler optimizes **self send** and **super send**; when the local dictionary count is greater than zero, the method compiler does not optimize **self send** and **super send**. The *int* argument and the **SetLocalDicts** call are removed from the method when the method is compiled.

SetLocalDicts can be used in two ways: to ensure that the method compiler optimizes **sends** when it should and to force the method compiler to optimize **sends** when it otherwise would not. An example of each case is given below.

If you define a method that leaves a local dictionary on the stack, you might cause the method compiler to optimize a **send** when it should not. The example below illustrates such a case. The following methods represent a portion of a class definition.

```
/method1 {
        /method2 self send
        /size self send
} def

/method2 {
        10 dict begin
                /size 1 def
                .
                .
} def

/size {
        .
        .
} def
```

In this example, /method2 puts a dictionary on the stack with a **begin**, but it does not remove the dictionary with an **end**. /method2 is invoked from within /method1. Therefore, a local dictionary is left on the stack in /method1, but the method compiler has no way to know that the local dictionary exists since its local dictionary counter is zero when it compiles /method1.

The method compiler optimizes the two **sends** in /method1 as follows:

```
/method1 {
        method2
        size
}
```

When /method1 is invoked, /method2 is called. /method2 puts a dictionary on the stack and defines a variable named /size. /method2 leaves the local dictionary on the stack. Then /size is encountered in /method1; /size is supposed to invoke the /size method, but since /size was just defined in the local dictionary that is still on the stack, /size refers to the variable instead of the method. Although this is a coincidence that the variable and method names are the same, the problem only occurred because /size self send was optimized by the method compiler.

You can use the **SetLocalDicts** directive to tell the method compiler to avoid optimizing /size self send, as follows:

```
/method1 {
        /method2 self send
        1 SetLocalDicts
        /size self send
} def
```

In this case, the local dictionary count is 1 when the method compiler reaches /size; therefore, /size self send is not be optimized. /method1 looks like the following after it is compiled:

```
/method1 {
        method2
        /size self send
} def
```

Although /method2 still leaves a local dictionary on the stack, the subsequent **send** removes the local dictionary before the /size method is executed.

In rare cases, you might want to leave a local dictionary on the stack before a **send**. The example code below illustrates how you could set the local dictionary count to be zero to force the method compiler to optimize two **self sends**.

```
/mymethod {
        10 dict begin
                0 SetLocalDicts
                /dothis self send
                /dothat self send
        end
} def
```

After the method compiler compiles this method, it looks like the following:

```
/mymethod {
        10 dict begin
                dothis
                dothat
        end
} def
```

When /mymethod is invoked, the two methods /dothis and /dothat are executed with the local dictionary on top of the stack.

4.6. Creating a New Instance

This section discusses the methods that NeWS provides to create and initialize instances. You send the **/new** message to create a new instance of a class. A class can use the standard object creation provided by **Object**'s **/newobject** method, or the class can alter the way an object is created. For example, the **/newmagic** method can be used to create a new instance from an existing NeWS magic dictionary. A class can initialize its instances with the **/newinit** method. To request the default implementation of a class, you can send the **/newdefault** message instead of the **/new** message (**/newdefault** is discussed in Section 4.7, *Intrinsic Classes*).

/new

<initializationargs> <creationargs> **/new** instance

Builds an instance of the class that receives the **/new** message. For example, the following expression creates a new instance of MyClass:

```
/new MyClass send
```

A class should not need to define its own **/new** method. Instead, the **/new** method in class **Object** is separated into two parts, and a class can choose to override either or both of the parts. These two parts are the two methods that **/new** calls: **/newobject** and **/newinit**. The **/newobject** method builds a new instance of a class, and the **/newinit** method initializes the instance.

When **/new** is sent to MyClass, the following steps are taken:

1. The **send** operator puts MyClass and its superclasses on the dictionary stack.
2. The **/new** method is located in **Object** (assuming no subclasses override **Object**'s **/new** method).
3. The **/new** method in class **Object** sends **/newobject** to MyClass to create a new instance of the class. The **/newobject** method leaves the newly created instance on the operand stack.
4. The **/new** method sends **/newinit** to the new instance to initialize it. A class' **/newinit** method adds anything that is unique to that class.
5. After invoking **/newobject** and **/newinit**, the **/new** method is done. The **/new** method leaves the new instance on the operand stack. The **send** operator takes MyClass and its superclasses off the dictionary stack to complete the **send**.

If a class requires arguments to its **/newobject** or **/newinit** methods, they must be passed to **/new** when an instance of the class is created. The following syntax creates an instance of MyClass and names the instance myinstance:

```
/myinstance <initializationargs> <creationargs> /new MyClass send def
```

The **/newobject** and **/newinit** methods are described in more detail below.

/newobject

<creationargs> **/newobject** instance

Creates an instance and leaves it on the operand stack. The **/newobject** method is called by **/new** when a new instance of a class is created. After calling **/newobject**, the **/new** method then calls **/newinit** to allow the class to initialize its new instance.

Class **Object**'s **/newobject** method creates an instance dictionary and copies the class' instance variables into it. The **/newobject** method also assigns an inheritance array to the instance.

Most classes do not need to override **/newobject**. The **/newmagic** method, discussed below, is an example of how a class might override the **/newobject** method.

/newinit

<initializationargs> **/newinit** —

Initializes a new instance. The **/new** method sends **/newinit** to the instance immediately after it has been created.

Class **Object**'s **/newinit** method performs no action. A class should provide its own **/newinit** method if it needs to initialize its instances. The **/newinit** method can perform any action that should be taken when a new instance of the class is created. If a class offers a **/newinit** method, the method should send **/newinit** to **super** to perform any initialization required by the class' superclasses, and then it should perform the class' initialization.

Below is an example of a class definition that uses the **/newinit** method. The class, called TimeKeep, is a subclass of class **Object**.

```
/TimeKeep Object
%instance variables:
dictbegin
  /Time null def
dictend
classbegin

%class variables:

  /ClassTime currenttime def

%methods:

  /newinit {
    /newinit super send
    /resettime self send
  } def

  /printtime {
    (Time is: ) print
    Time 10 string cvs print
    (\n) print
  } def

  /resettime {
    /Time currenttime def
  } def

 classend def
```

Class TimeKeep has a class variable, ClassTime, that is set to the time of creation of the class. TimeKeep has an instance variable named Time. Class

TimeKeep's /**newinit** method first sends /**newinit** to **super**; then it calls the /resettime method to initialize the instance variable Time to be the time of creation of the instance (the time at which the method is called). The method printtime prints the value of the instance variable Time.

The following expression defines an instance of class TimeKeep named timer:

```
/timer /new TimeKeep send def
```

The expression below prints the value of timer's instance variable Time:

```
/printtime timer send
```

A class' instance variables can often be initialized in a dictionary passed to **classbegin**; usually, you do not need to use **newinit** to assign initial values to instance variables. However, you can use /**newinit** to make the initialization of instance variables more efficient.

When you create a new instance, the /**newobject** method copies all the class' instance variables into the new instance dictionary. This copying takes less time for simple instance variables than for composite instance variables. Therefore, whenever you can avoid declaring a composite instance variable in a dictionary passed to **classbegin**, you shorten the amount of time required to create a new instance of that class. This time difference is more significant if you can arrange your class definition to avoid passing any composite instance variables to **classbegin**. To initialize a null dictionary, for example, you might define a simple instance variable to be null in the dictionary that you pass to **classbegin** and then define that variable to be a **growabledict** in a /**newinit** method for the class. This arrangement is faster than simply defining the variable to be **nulldict** in the dictionary that you pass to **classbegin**.

Note that you can pass composite instance variables to **classbegin** when necessary; your code is just more efficient if you minimize the number of composite instance variables passed to **classbegin** in your class definitions.

/**newmagic**

<creationargs> dict /**newmagic** instance

Builds an instance from an existing NeWS dictionary object such as a canvas or an event. To create such an instance, you send the /**new** message to the desired class of the object, and the class overrides the /**newobject** method with the /**newmagic** method.

The /**newmagic** method takes a magic dictionary object from the stack and uses the key-value pairs in the magic dictionary as instance variables. The instance is also given any instance variables specified by its class. The magic dictionary is turned into an instance dictionary by adding the additional instance keys; this is possible because, by definition, a magic dictionary can have keys added to it.

Suppose you have a class called Canvas that is used to create instances that are canvas objects. You could define class Canvas in the following way:

```
/Canvas Object
dictbegin
        %instance variables
        .
        .
dictend
classbegin

        %class variables
        .
        .

        %class methods

        /newobject {
                newcanvas
                /newmagic super send
        } def

        /newinit {
                %initialize canvas instance variables
                .
                .
        } def

        .
        .
classend
```

You could create an instance of class Canvas by sending the **/new** message to class Canvas. When you do this, the /new method in class **Object** sends **/newobject** to **self**, and class Canvas overrides the **/newobject** method with its own version. Canvas' **/newobject** method calls the canvas operator **newcanvas** to create a new, empty canvas dictionary. Then Canvas' **/newobject** method calls **/newmagic** to make an instance dictionary out of the canvas dictionary.

Note that an instance of Canvas is a true NeWS canvas. For example, if you change the **Mapped** instance variable from false to true, the canvas will be mapped to the screen. The canvas is part of the canvas hierarchy, but the instance and class Canvas are part of the class hierarchy.

4.7. Intrinsic Classes

Sometimes you want a class to be a common, abstract superclass for a group of subclasses. An abstract superclass provides an easy way to implement many different versions of the object that the superclass represents. The abstract superclass defines a set of basic characteristics that all its subclasses must have, but the superclass allows many of the implementation details to vary from subclass to subclass. In fact, an abstract superclass can demand that its subclasses implement certain methods that it does not implement itself. Usually, an abstract superclass does not have direct instances; instead, its subclasses have instances. In NeWS, abstract superclasses are known as *intrinsic* classes.

For example, Window could be an intrinsic class that implements different types of windows. Each subclass of Window might implement a different "look and feel" for the window's user interface.

An intrinsic class should specify a default subclass; then if the **/newdefault** message is sent to the intrinsic class, the newly created instance belongs to that default subclass (see **/newdefault** below).

The three methods described below are often used with intrinsic classes.

/newdefault

<initializationargs> <creationargs> **/newdefault** instance

Creates a new instance of a class' default implementation by sending the **/new** message to the class' default subclass. If a class has no default subclass, the server assumes that the default implementation is the class itself.

The following expression creates a new instance of the default subclass of Window:

```
/newdefault Window send
```

For example, if the default subclass of Window is MyWindow, the above expression causes **/new** to be sent to MyWindow.

A class' default subclass is specified by a class variable named **DefaultClass**. You can set the value of **DefaultClass** in the class definition. The example below sets the default class for Window to be MyWindow. Note that the value of the **DefaultClass** variable is the default subclass inside procedure braces; the braces are needed to defer execution until the default subclass is defined.

```
/Window [Canvas]
instance variables
classbegin

        /DefaultClass {MyWindow} def
        .
        .
classend
```

A user can override the default implementation of a class by including a procedure in the **UserProfile** dictionary (see Section 4.8, *Overriding Class Variables With UserProfile*).

/defaultclass

— **/defaultclass** class

Returns the default subclass of the class that receives the **/defaultclass** message. The default subclass is specified by a class' **DefaultClass** variable. If a class has no **DefaultClass** variable, the default implementation is the class itself.

/SubClassResponsibility

— **/SubClassResponsibility** —

Requires a subclass to implement a certain method. **/SubClassResponsibility** causes a deliberate `undefined` error if the required method is sent to a subclass that does not implement it.

For example, the method /CreateFrameMenu must be implemented by any subclass of Window if Window has the following code in its class definition:

```
/CreateFrameMenu {SubClassResponsibility} def
```

If the message /CreateFrameMenu is sent to a subclass of Window that does not implement the /CreateFrameMenu method, **/SubClassResponsibility** causes an `undefined` error.

4.8. Overriding Class Variables With UserProfile

UserProfile is a dictionary in `.startup.ps` that contains user-supplied information. A user can add procedures to **UserProfile** to override the default values of class variables. (See the *X11/NeWS Server Guide* for more information about **UserProfile**.)

The **classend** operator completes the definition of a class. The last step that the **classend** operator takes is to check the **UserProfile** dictionary for a procedure with the same name as the class that is currently being defined. If the **classend** operator finds such a procedure, it executes the procedure with the class name and the class object on the stack. The procedure must leave the stack unchanged.

The following example shows part of a **UserProfile** dictionary. In this example, the procedure named Frame overrides the default value of FrameColor for class Frame; the procedure sets the value of FrameColor to be gray.

```
UserProfile begin
        .
        .
        /Frame {   %classname class => classname class
                dup /FrameColor .75 .75 .75 rgbcolor put
        } def
        .
        .
end
```

Overriding DefaultClass

A user can include a procedure in **UserProfile** that assigns a new value to a class' **DefaultClass** variable; the new value overrides the value assigned in the class definition. (For an explanation of **DefaultClass** see **/newdefault** in Section 4.7, *Intrinsic Classes*).

Assume that the default class of Window is set to MyWindow by the programmer (in the class definition). If a user wants the default implementation of class Window to be SpecialWindow instead of MyWindow, the user could add the following definition to the **UserProfile** dictionary:

```
UserProfile begin
        .
        .
        /Window {    %classname class => classname class
                dup /DefaultClass {SpecialWindow} put
        } def
        .
        .
end
```

Note that SpecialWindow must be given in braces.

4.9. Promoting Class Variables to Instance Variables

An instance can override a class variable by *promoting* that class variable to be an instance variable. Class **Object** provides utilities to promote a class variable to an instance variable and to inquire about the current promotion status of a variable. These utilities are described below.

promote

name value **promote** —

Takes a name and a value from the operand stack and adds that name-value pair to the dictionary that is on top of the dictionary stack, exactly as the **def** operator does. The **promote** utility is called when an instance dictionary is on top of the stack so that the name-value pair becomes an instance variable. The **promote** utility is just a formal way to use **def** instead of **store**; you should use **promote** instead of **def** because **promote** makes your intention clear.

Suppose you have a class named Frame and an instance of the class named myframe. (A frame is an object that "frames" a canvas. The frame might offer such features as a menu and scrollbars.) Assume that one of Frame's class variables is FrameColor, which is the color of the frame's background. Also assume that the default color of FrameColor is white. You can give myframe a gray FrameColor by putting myframe's dictionary on top of the stack and then promoting the class variable FrameColor as follows:

```
/FrameColor .75 .75 .75 rgbcolor promote
```

In the above example, **promote** adds FrameColor to myframe's instance variable dictionary and assigns the value returned by the **rgbcolor** operator to the new instance variable.

unpromote

name **unpromote** —

Removes, or *unpromotes*, an instance variable from the instance's dictionary. The **unpromote** utility takes the name of the variable from the operand stack and removes that variable from the dictionary that is on top of the dictionary stack. After putting myframe on top of the stack, you could remove FrameColor from myframe's dictionary with the following expression:

```
/FrameColor unpromote
```

promoted?

name **promoted?** boolean

Takes the name of a variable from the operand stack and returns true if that variable is found in the dictionary that is on top of the stack. Assuming that myframe is on top of the dictionary stack, the following example returns true if FrameColor is an instance variable (and was therefore promoted):

```
/FrameColor promoted?
```

Avoiding an Accidental Promotion

If you try to use a **def** statement to change the value of a class variable while an instance is on the top of the dictionary stack, you will add that variable to the instance, effectively promoting it. If you just want to change the value of the class variable, you should use **store** instead of **def**. The **store** operator finds the first occurrence of the variable on the dictionary stack and replaces the value of the variable with the newly specified value. (The **def** operator adds the name-value pair to the top dictionary on the stack if it does not find the variable already in that dictionary.)

This accidental promotion can occur even if you use **def** in a method that changes the value of the class variable because the method might be sent to an instance of the class, putting the instance dictionary on top of the stack. To be safe, you should always use **store** to define values of class variables.

4.10. Destroying Classes and Instances

Instances are destroyed with the **/destroy** method; classes are destroyed with the **classdestroy** operator. The **classdestroy** operator invokes a utility named **/cleanoutclass**. The **/destroy** method, **classdestroy** operator, and **/cleanoutclass** method are described below.

/destroy

— **/destroy** —

Destroys the instance that receives the **/destroy** message. An application might invoke **/destroy** when a user chooses the "quit" option from a menu. Classes should provide their own **/destroy** methods. A class' **/destroy** method should remove circular references and then send **/destroy** to **super**. The **/destroy** method in class **Object** performs no action; it is just there so that classes can safely send **/destroy** to **super**.

classdestroy

class **classdestroy** —

Destroys a class. **classdestroy** removes several circular references to the class by removing the class from the subclass lists of its superclasses. Then **classdestroy** sends the **/cleanoutclass** method (see description below) to the class.

/cleanoutclass

— **/cleanoutclass** —

Calls the **cleanoutdict** operator, which is a NeWS utility that undefines every key in the specified dictionary using the **undef** NeWS primitive. The **/cleanoutclass** method is defined by the **classbegin** operator. A class can override the default **/cleanoutclass** with its own clean-up procedure, if necessary.

4.11. Obsolete Objects in the Class System

When all the references to an object are *soft*, the object is *obsolete* and NeWS sends an *obsolescence* event to all processes that have expressed interest in obsolescence events for that object (see Chapter 7, *Memory Management*). The processes should then remove their soft references to the object so that NeWS can destroy the object and reclaim the memory that it used.

When class **Object** is initialized, it starts a process named **ObsoleteEventMgr** that expresses interest in obsolescence events. When **ObsoleteEventMgr** receives an obsolescence event from the server, it invokes a method in class **Object** that handles obsolescence events. This method performs the following actions:

- If the obsolescence event is for a class, the **classdestroy** operator is called to destroy the class (see **classdestroy** in Section 4.10, *Destroying Classes and Instances*).
- If the obsolescence event is for an instance, the **/obsolete** method is sent to the instance to destroy it (see the description of **/obsolete**, below).
- If the obsolescence event is not for a class or an instance, it is simply popped from the stack.

The **/obsolete** method is described below.

/obsolete

— **/obsolete** —

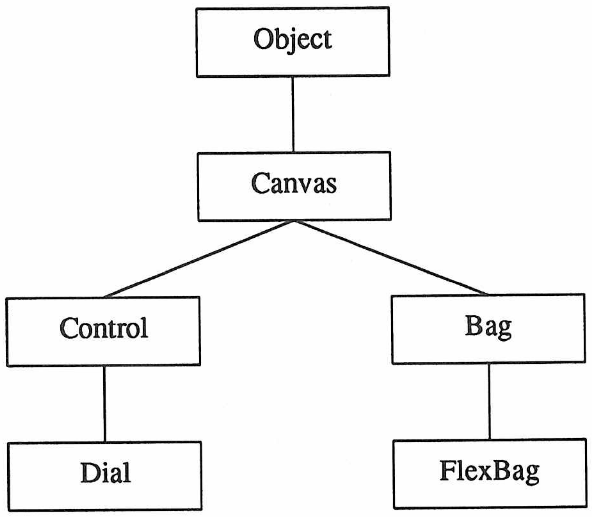
Sends **/destroy** to **self** (see the explanation of **/destroy** in Section 4.10, *Destroying Classes and Instances*). When class **Object**'s **ObsoleteEventMgr** receives an obsolescence event for an instance, the **/obsolete** message is sent to the instance to destroy it. A class rarely needs to override the default **/obsolete** method.

Note that instances are usually destroyed without having to call **/obsolete**; the **/destroy** method is usually called directly to destroy an instance.

4.12. Multiple Inheritance

Multiple inheritance is an optional aspect of the NeWS class system. You can build a whole class tree without using multiple inheritance. However, in some situations, multiple inheritance is very useful and easy to apply. This section first gives an example of a simple case to illustrate why you might want to use multiple inheritance, and then it gives a more complex example to explain the details of multiple inheritance. Both the simple example and the more complex example use the class structure shown in the following figure:

Figure 4-7 *Basic class hierarchy for the multiple inheritance examples*



Class Canvas is a subclass of class **Object**. In this example, class Canvas has two immediate subclasses: Control and Bag. Control represents a type of canvas that handles user interaction for objects such as buttons and dials. Bag represents a special type of canvas that contains objects; an instance of Bag can perform layout and intelligent repainting of its contained objects.

Control and Bag each have one subclass. Control has a subclass named Dial that provides basic operations needed by sliders and scrollbars. Bag has a subclass named FlexBag; an instance of FlexBag can arrange its contained objects by specifying interobject relationships based on compass directions.

So far, each class in this tree only specifies one immediate superclass. For example, Dial's immediate superclass is Control, and FlexBag's immediate superclass is Bag.

A Simple Multiple Inheritance Example: a Utility Class

For convenience and efficiency, you can define a utility class that contains low-level methods needed by many of your classes. You can define a utility class that exists apart from the main class tree — a class with no superclasses. To create such a class, you specify an empty superclass array to the **classbegin** operator, as follows:

```
/Utility [ ]
instance variables
classbegin
        class variables
        class methods
classend def
```

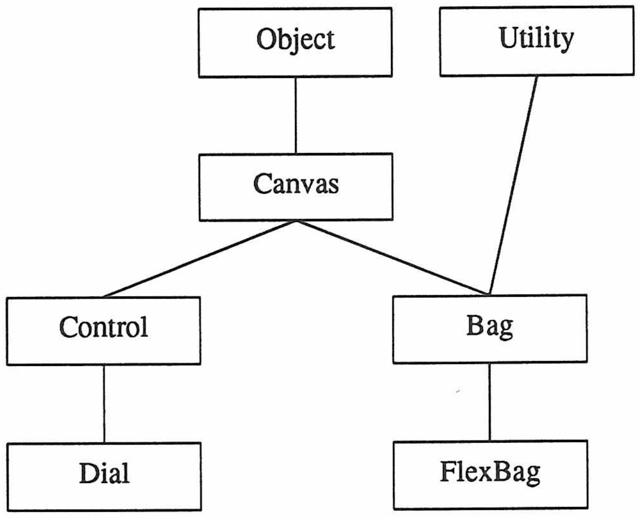
In fact, this is how class **Object** is defined. But class **Object** is the root of the class tree, whereas class Utility is a utility class. Multiple inheritance allows the classes in the main class tree to access class Utility's methods.

Assume that you want class Bag to be able to access the methods in Utility. When you create class Bag, you could specify both Utility and Canvas in the superclass array that you give to **classbegin**. For example, your class definition could take the following form:

```
/Bag [Utility Canvas]
instance variables
classbegin
        class variables
        class methods
classend def
```

The class tree is illustrated below. Note that Bag now has two immediate superclasses; therefore two lines connect it to classes above it.

Figure 4-8 *Class hierarchy with a utility class*



A class' superclasses include the class' immediate superclasses and all of their superclasses. As shown in the diagram of the class tree, Bag's superclasses are Utility, Canvas, and **Object**. Although the tree does indicate which classes are Bag's superclasses, it does not indicate a unique order in which the superclasses should override each other. The superclasses do not belong to the same branch, so a unique leaf-to-root order is no longer possible.

Thus in the multiple inheritance case, more than one valid order exists for the classes in an inheritance array. A valid array consists of any arrangement of the superclasses that maintains the leaf-to-root order of classes on the same branch. Based on its superclasses, the three valid arrays for Bag in this example are the following:

[Utility Canvas Object]

[Canvas Utility Object]

[Canvas Object Utility]

In some situations the order does not matter. If the classes in the inheritance array have no methods or class variables in common, the order of those classes makes no difference to the final result of a **send**.

With a utility class, all that matters is whether any classes override the methods in the utility class. If the classes in the main class tree do not override any of the utility class' methods, you can place the utility class anywhere in the inheritance array and the results will be the same.

If you only specify a class' immediate superclasses in the array that you pass to **classbegin**, the **classend** operator uses an algorithm to construct a default order for the inheritance array. The **classend** operator starts with a copy of the superclasses that you pass to **classbegin**, and it adds the other superclasses to build the complete inheritance array. After your new class is created, you can examine the default order of the inheritance array by sending the **/superclasses** method to the new class. The **/superclasses** method puts the inheritance array on the operand stack (see Section 4.15, *Utilities for Inquiring About an Object's Heritage*).

If you do not like the default order of the inheritance array, you can change your class definition to achieve the order you want. You can alter the order of the inheritance array by changing the order of the superclasses that you give to **classbegin** or by listing more superclasses in the array that you give to **classbegin**. You can even list every superclass in the array that you give to **classbegin** so that the inheritance array will be exactly what you specify. These options are explained in more detail in the subsection *A More Complex Multiple Inheritance Example*, below.

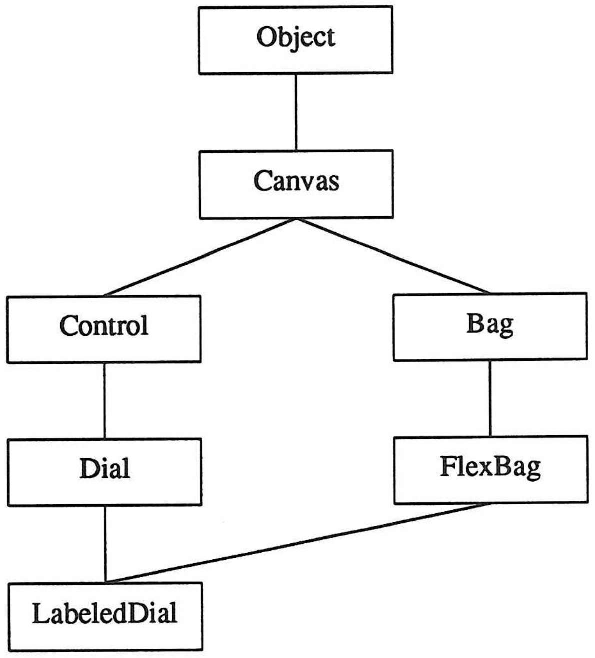
Once the inheritance array is constructed, the class mechanism works in the same way for multiple inheritance as it does for single inheritance. If a message is sent to Bag, any existing **send** context is temporarily removed, and then the classes in Bag's inheritance array are placed on the dictionary stack. In this way, **sends** to Bag can locate methods that reside in the utility class. Multiple inheritance does not affect how the **send** operator works; it just determines the inheritance array that **send** puts on the stack.

A More Complex Multiple Inheritance Example

This example shows how to use multiple inheritance to create a subclass of Dial named LabeledDial. The new type of dial has the basic characteristics of Dial, and it also has the capabilities of FlexBag: an instance of LabeledDial is a dial that can place a label north, south, east, or west of the dial itself. LabeledDial inherits from both Dial and FlexBag because both these classes are specified in the superclass array that is given to **classbegin**.

To simplify this example, the utility class is omitted. The following figure illustrates the class tree. Note that class LabeledDial has two immediate superclasses: Dial and FlexBag.

Figure 4-9 *Class tree for* LabeledDial *example*



Again, the class tree does not indicate the order of the superclasses in LabeledDial's inheritance array, but it does indicate which classes belong in the inheritance array: Dial, FlexBag, Control, Bag, Canvas, and **Object**. (The superclasses of LabeledDial include LabeledDial's immediate superclasses and all of their superclasses.)

Rules for Valid Inheritance Array Orders

The basic rules for valid inheritance arrays in the NeWS class system are given below:

(1) Classes on the same branch of the tree must be listed in leaf-to-root order in the inheritance array.

(2) If class A precedes class B in the superclass array that is passed to **classbegin** for class C, then class A must precede class B in the inheritance array of class C.

(3) If class A precedes class B in the inheritance array of a superclass of class C, then class A must precede class B in the inheritance array of class C.

Possible Inheritance Arrays for this Example

Given the above rules, more than one valid inheritance array is possible for LabeledDial.

Assume that the following array is given to the **classbegin** operator for Labeled-Dial:

[Dial FlexBag]

Based on the **classbegin** superclass array above (and the previously-defined rules), the following two arrays are valid inheritance arrays for LabeledDial:

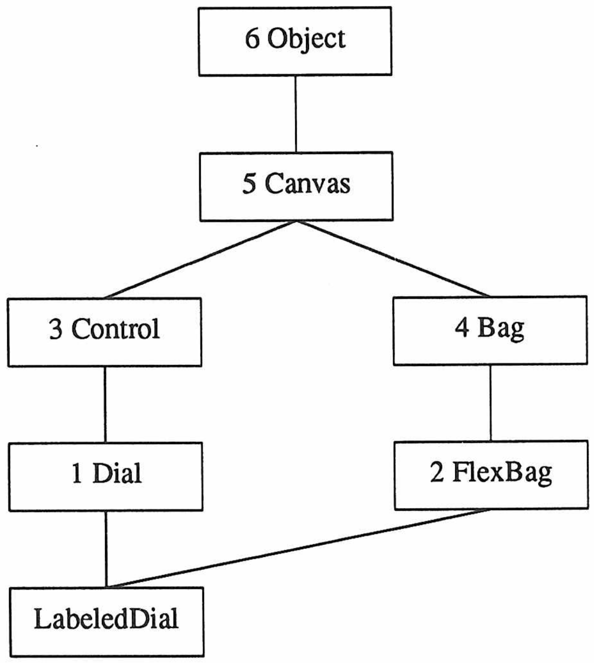
[Dial FlexBag Control Bag Canvas Object] (A)

[Dial Control FlexBag Bag Canvas Object] (B)

Note that the first array is a leaf-to-root *breadth-first* search through the tree and the second array is a leaf-to-root *depth-first* search through the tree. The breadth-first search moves up the tree one level at a time; classes at one level of the tree are included in the array before the search moves up to the next level of the tree. The depth-first search follows each branch, in turn, until the point at which the branch meets the next branch; classes on one branch are included in the array before the search moves down the tree to start again with the next branch. Both these search types start with the first class listed in the **classbegin** superclass array, and both search types satisfy the server's rules for valid inheritance arrays.

The following picture illustrates the breadth-first order (inheritance array (A) given above). Each class name has a number before it that indicates that class' position in the inheritance array.

Figure 4-10 *A breadth-first order for* LabeledDial*'s inheritance array*



The following picture illustrates the depth-first order (inheritance array (B) given above). Each class name has a number before it that indicates that class' position in the inheritance array.

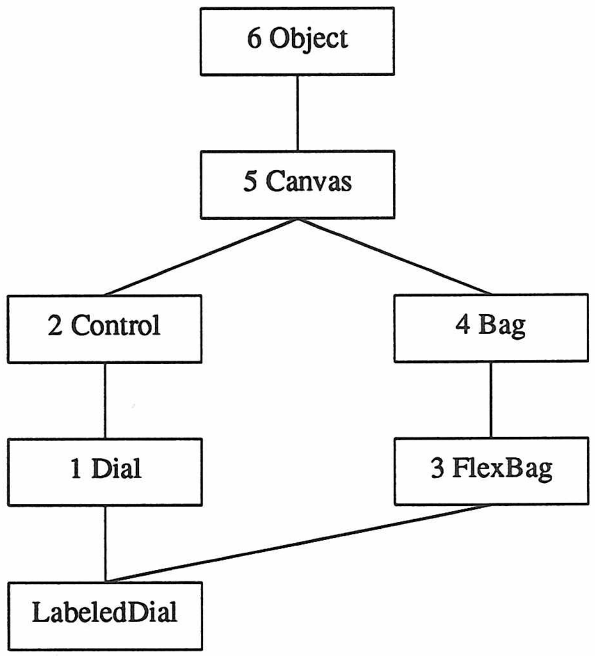


Figure 4-11 *A depth-first order for* LabeledDial*'s inheritance array*

Assume that the order of the superclasses in the superclass array passed to **classbegin** is reversed, as follows:

[FlexBag Dial]

Based on the above **classbegin** superclass array (and the previously-defined rules), the following two arrays are valid inheritance arrays for LabeledDial:

[FlexBag Dial Bag Control Canvas Object]

[FlexBag Bag Dial Control Canvas Object]

Again, one order represents a breadth-first search and one order represents a depth-first search. In this case, the inheritance arrays begin with FlexBag instead of Dial because FlexBag is listed before Dial in this version of LabeledDial's **classbegin** superclass array. The order of the classes in the **classbegin** superclass array is always maintained in the inheritance array (rule 2 in the subsection *Rules for Order of Inheritance Arrays*, above).

Which Order Do You Choose?

You choose the order of the inheritance array based on the order in which you want the classes to override each other. If it makes no difference, you can specify just the two immediate superclasses and let the server create the default array based on your **classbegin** superclass array. To examine the default order, you can send the /**superclasses** method to the newly created class (see Section 4.15, *Utilities for Inquiring About an Object's Heritage*).

Constraining the Order of the Inheritance Array

If you do not like the default order, you can constrain the order of the classes in the inheritance array by specifying more classes in the **classbegin** superclass array. Note that you must always list the superclasses in leaf-to-root order.

Assume that LabeledDial's **classbegin** superclass array is specified as follows:

[Dial Control FlexBag]

Based on the above **classbegin** superclass array, the inheritance array for LabeledDial is the following:

[Dial Control FlexBag Bag Canvas Object]

In effect, you have forced the inheritance array to be the depth-first choice that starts with Dial.

You could force the array to be the breadth-first choice that starts with Dial by specifying the **classbegin** superclass array as follows:

[Dial FlexBag Control]

In this case, the inheritance array for LabeledDial is the following:

[Dial FlexBag Control Bag Canvas Object]

All you are doing is specifying more of the array to achieve the order you desire. The extreme case is to list the entire inheritance array that you desire. If you list every superclass in the **classbegin** superclass array, and if you give a valid order, the inheritance array will be identical to the superclass array that you specify.

super and Multiple Inheritance

With multiple inheritance, the **send** operator still puts the classes in the message receiver's inheritance array on the dictionary stack and searches for the specified method. The **super** psuedo-variable still refers to the superclass that follows the method's class in the current **send** context, but note that **super** could mean different things to different classes.

For example, suppose that class Control has a method named /method1 that sends /method2 to **super**. Also suppose that /method1 is not overridden by any classes beneath Control in the class tree. In this example, /method1 has the following structure:

```
/method1 {
        .
        .
        /method2 super send
        .
        .
} def
```

As illustrated in the previous diagram of the class tree, Control's inheritance array is the following:

[Canvas Object]

If /method1 is sent to Control, Control and the superclasses in Control's inheritance array are put on the dictionary stack. Then /method1 is located in Control, and the **send** to **super** is encountered. The **super** pseudo-variable refers to the class that is below /method1's class in the current **send** context; in this case, the **super** in /method1 refers to Canvas.

Assume that LabeledDial's inheritance array is the following:

[Dial FlexBag Control Bag Canvas Object]

If /method1 is sent to LabeledDial, LabeledDial and the superclasses in LabeledDial's inheritance array are put on the dictionary stack. Then /method1 is found in Control, and the **send** to **super** is encountered. The **super** pseudo-variable still refers to the class that is below /method1's class in the current **send** context, but in this case, that class is Bag. Therefore, if /method1 is sent to LabeledDial, the search for /method2 starts with Bag.

The **super** psuedo-variable is always evaluated within the current context. Therefore, the **super** in Control's method refers to Canvas if Control's inheritance array is on the stack, but it refers to Bag if LabeledDial's inheritance array is on the stack.

4.13. Utilities for Setting and Retrieving an Object's Name and Classname

Each class has a **ClassName** variable that is assigned the *classname* that you pass to the **classbegin** operator. In addition to the **ClassName** variable, each class also has a **Name** variable. The default value of the **Name** is the **ClassName**. You can set the value of the **Name** variable to something other than **ClassName**; this is generally done for an instance by promoting **Name** to be an instance variable and then giving the instance a name.

The class methods that set and retrieve the values of **Name** and **Classname** are described below.

/name

— **/name** name

Returns the name of the object that receives the **/name** message. An object's name is stored in its **Name** variable. The **Name** variable defaults to the **ClassName**.

/setname

name **/setname** —

Assigns the specified *name* to the **Name** variable of the object that receives the **/setname** message. If you send this message to an instance, the **Name** variable is promoted to an instance variable. The following example promotes **Name** to be an instance variable for MyInstance and sets the value of **Name** to be /MyInstance:

```
/MyInstance /setname MyInstance send
```

/classname

— **/classname** name

Returns the class name of the class that receives the **/classname** message. The class name is stored in a class' **ClassName** variable. The **ClassName** variable defaults to the *classname* that you pass to **classbegin**.

4.14. Utilities for Inquiring About an Object's Status

You can use the operators described in this section to inquire about the status of an object. You can ask whether the object is a "sendable object" (an instance or a class), whether the object is a class, or whether the object is an instance.

isobject?

object **isobject?** boolean

Takes *object* from the top of the operand stack and returns true if the object is an instance or a class. Returns false if the object is not an instance or a class.

isclass?

object **isclass?** boolean

Takes *object* from the top of the operand stack and returns true if the object is a class or false if the object is not a class.

isinstance?

object **isinstance?** boolean

Takes *object* from the top of the operand stack and returns true if the object is an instance or false if the object is not an instance.

4.15. Utilities for Inquiring About an Object's Heritage

You can use the methods described in this section to inquire about an object's heritage and to retrieve information concerning the object's relationship to other objects.

/superclasses

— **/superclasses** array

Returns the inheritance array of the object that receives the **/superclasses** message.

/subclasses

— **/subclasses** array

Returns the subclass array of the class that receives the **/subclasses** message. Class B is in the subclass array of class A if class A was given to the **classbegin** operator as a superclass of class B.

/instanceof?

object **/instanceof?** boolean

When the **/instanceof?** message is sent to a class, the method takes the top *object* off the operand stack and returns true if the object is an instance of the class or false if it is not.

/descendantof?

object **/descendantof?** boolean

When the **/descendantof?** message is sent to a class, the method takes the top *object* off the operand stack and returns true if the class is in the object's inheritance array.

/understands?

name **/understands?** boolean

When the **/understands?** message is sent to an object, the method takes the specified *name* off the operand stack and returns true if any of the classes in the object's inheritance array has a method with that specified *name*.

/class

— **/class** class

When the **/class** message is sent to an instance, the method returns the instance's class.

4.16. Utilities for Finding Objects on the send Stack

The **send** *stack* is a record of all the **send** contexts that have accumulated during a nested **send**. The **send** stack is not the same as the dictionary stack; the dictionary stack only contains the current **send** context, but the **send** stack contains all the **send** contexts that came before the current **send** context in a nested **send**. The **send** stack is aranged with the oldest context on the bottom and the most recent context on the top.

You can use the utilities described in this section to locate the top instance or top descendant of a class on the **send** stack or to send a message to the top descendant on the **send** stack.

/topmostinstance

— **/topmostinstance** object *or* null

When the **/topmostinstance** message is sent to a class, the method finds and returns the class' topmost instance on the **send** stack; if no such instance exists, the method returns null.

/topmostdescendant

— **/topmostdescendant** object *or* null

When the **/topmostdescendant** message is sent to a class, the method finds and returns the class' topmost descendant on the **send** stack; if no such object exists, the method returns null. A class' descendant is defined as an object that has that class in its inheritance array.

/sendtopmost

<args> name **/sendtopmost** <results>

When **/sendtopmost** is sent to a class, it **sends** *name* to the topmost descendant of the class (see the explanation of **/topmostdescendant**, above). If *name* requires arguments, they should be specified.

4.17. Class Operators

classname superclasses instvars	**classbegin**	–
class	**classdestroy**	–
–	**classend**	classname newclass
object	**isobject?**	boolean
object	**isclass?**	boolean
object	**isinstance?**	boolean
name object	**promote**	–
name	**promoted?**	boolean
name object	**redef**	–
name	**unpromote**	–
–	**self**	object
<args> name object	**send**	<results>
<args> proc object	**send**	<results>

4.18. Class Methods

–	**/class**	class
–	**/classname**	name
–	**/cleanoutclass**	–
–	**/defaultclass**	class
object	**/descendantof?**	boolean
–	**/destroy**	–
<args> proc	**/doit**	<results>
name proc	**/installmethod**	–
object	**/instanceof?**	boolean
uncompiledproc	**/methodcompile**	compiledproc
–	**/name**	name
<args>	**/new**	instance
<args>	**/newdefault**	instance
<args>	**/newinit**	–
<args> dict	**/newmagic**	instance
<args>	**/newobject**	instance
–	**/obsolete**	–
<args> name	**/sendtopmost**	<results>
–	**/superclasses**	array
–	**/subclasses**	array
–	**/topmostdescendant**	null *or* object
–	**/topmostinstance**	null *or* object
name	**/understands?**	boolean
name	**/setname**	–
–	**/SubClassResponsibility**	–

5

Client-Server Interface

5 Client-Server Interface

The X11/NeWS server contains facilities for communicating with client programs that run either locally or remotely. Each client is permitted to send POSTSCRIPT language code to the server. The server runs this code on behalf of the client.

Typically, a client program contains two main sections; one, which can be written in C, FORTRAN, or any other language, is used to perform the basic computations that are required; the other, which must be written in the POSTSCRIPT language, is used to provide corresponding windows or graphics. The POSTSCRIPT language section of the client program can be detached, sent to the server, and executed remotely by means of function calls.

This form of interprocess communication has the advantage of allowing great freedom of execution for the respective parts of a process. The POSTSCRIPT language code downloaded by the client program can reference any of NeWS' inbuilt features, including procedures defined in the **userdict** and **systemdict** dictionaries. The server contains no predefined messages with which to respond to the client; for example, no routine exists for notifying the server when the user manipulates the mouse. Instead, the way in which the client and server communicate is specified entirely by the programmer in terms of the contents of the client application.

Most programmers are likely to use C as the language of the client application. Therefore, NeWS provides a special interface facility that supports C-client communication (most of this chapter is concerned with this facility). However, it is also possible for programmers to create their own interface facility for use with other languages; information is provided on how this can be achieved.

NOTE *Users who wish to download pure POSTSCRIPT programs to the server should use the program* `psh(1)`.

5.1. The CPS Facility

The *C to POSTSCRIPT* facility (*CPS*) provides an interface that allows C client applications to communicate with the server. The facility allows a client program to define POSTSCRIPT language routines and associate them with C function-names; these functions and their corresponding arguments can then be downloaded into the server and executed. The facility provides functions that open and close server communication, utilities that implement commonly used POSTSCRIPT language operators, and a token-definition system that allows data to be compressed before it is sent to the server.

To use the CPS facility, a client application must contain three files, which are summarized as follows:

- A `.cps` file

  This file contains POSTSCRIPT language expressions to be executed within the server; each expression is specified as an argument to the `cdef` command, which associates the expressions with a C function-name and other facilities required for client-server communication.

- A `.h` header file

  This file is created by specifying a defined `.cps` file as the argument to the `cps` command; the `.h` file contains the POSTSCRIPT language expressions from the `.cps` file as specially compiled macros that are comprehensible to the C compiler.

  A `.h` header file automatically includes the file `<NeWS/psmacros.h>`, which contains definitions of standard CPS macros and function-declarations residing in `libcps.a` (this library must be added to the list of libraries searched by the linker, using `-lcps` on the compile command line); `<NeWS/psmacros.h>` also contains `#include` statements for both the standard I/O package `<stdio.h>` and the NeWS I/O package `<NeWS/psio.h>`.

- A `.c` file

  This file typically contains the main section of the C client program. The file must begin with an `#include` statement that includes the previously created `.h` file. The C function-names defined in the `.cps` file can then be referenced freely within the `.c` file and thereby used within the server to call the associated compiled POSTSCRIPT language expressions.

  NeWS provides C functions for opening, maintaining, and closing a line of communication between the client and the server. These functions may also be used freely within the `.c` file.

These three files are described in detail throughout the following sections.

5.2. Creating the `.cps` File

The `.cps` file must consist of `cdef` statements, each of which defines a macro to be used in server communication. The `cdef` command can be used to specify both of the following:

- POSTSCRIPT language code to be sent to the server for execution
- Results returned from the server after execution

The full syntax of the `cdef` command is as follows:

`cdef` *name*`(`*args*`)` `=>` *tag* `(`*results*`)` *POSTSCRIPT-code*

- *name*

  This is the name of the macro as it appears in the client program

- *args*

  This represents any number of arguments to be passed to the C macro defined by `cdef`. Each argument can be specified either as a value to be used in the specified POSTSCRIPT language computation or as a pointer-variable combination into which a result is read when returned from the server. Note that the specified *args* must come immediately after the specified *name*.

- =>

  These symbols are used to indicate that the following integer (the *tag*) and parenthesized list (the *results*) are the specification of a packet to be received by the client when it executes this macro.

- *tag*

  This is an identifier associated with the specified *results*. The identifier is used to prevent confusion when multiple NeWS processes are simultaneously writing results back to the client. The identifier must be a unique integer constant or must appear in the list of arguments to the `cdef` as an integer argument.

- *results*

  This is an optional list of one or more variables that receive the values returned from the server's execution of the specified POSTSCRIPT language code. Each variable must also be included in the *args* argument.

  Note that the =>, *tag*, and *results* arguments must come together and must appear after the *name* and (*args*) arguments: however, they can appear before, after, or in the middle of the specified *POSTSCRIPT-code*.

- *POSTSCRIPT-code*

  This is the POSTSCRIPT language procedure invoked within the server when the *name* macro is called. The POSTSCRIPT language code can continue for several lines: indentation is not important. One `cdef` statement is always terminated by the start of another.

The *args*, =>, *tag*, *results*, and POSTSCRIPT-code arguments are optional. Thus, a `cdef` statement may simply specify POSTSCRIPT language code to be executed without arguments and without the need to returns results, or it can specify the contents of a return packet with no POSTSCRIPT code to be sent to the server.

If =>, *tag*, and *results* arguments are specified, the server must be made to return values from its execution of some POSTSCRIPT language code. NeWS provides two operators, **tagprint** and **typedprint**, that do this. The values are returned as a *packet* of data, in which they are preceded by the specified *tag*. The tag thus separates its own packet from any others that might appear in the data stream flowing from the server to the client.

Thus, the uses of the `cdef` statement can be classified as follows:

- Sending POSTSCRIPT language code to the server without requesting that values be returned.

  This POSTSCRIPT language code may send packets of data back to the client for retrieval by some other `cdef` statement.

- Sending POSTSCRIPT language code to the server, explicitly requesting that a given set of results be returned, and blocking until this has occurred (this is termed *receiving a synchronous reply*).

- Sending no POSTSCRIPT language code to the server, explicitly requesting that a given set of results be returned, but continuing to run without blocking if a different set of results has been sent back by POSTSCRIPT language code running in the server (this is termed *receiving an asynchronous reply*).

These three uses are described in detail throughout this chapter.

Argument Types

Each parameter specified in the *args* field of the `cdef` command must have a specified type; the default type is `int`. To specify a type, precede the parameter with the appropriate type name. The syntax is thus as follows:

```
cdef name(type-name arg1, type-name arg2) POSTSCRIPT-code
```

NOTE *`int` uses the natural data type size for a computer; this is 32 bits for a Sun; however, in the current release, `int`s are sent over the network as* **16-bit** *integers. You need to be aware of this to avoid portability problems with some other machines.*

Most of the types correspond directly to C types. The following table lists the CPS argument types:

Table 5-1 *CPS Argument Types*

CPS type	*C type*
`int`	`int`, `long`, and `char` (this is the default type in `cdef` specifications)
`float`	`float` or `double`
`string`	`char *` strings that are null terminated
`cstring`	`char *` with an accompanying count of the number of characters in the string. Counted strings have two arguments in the C function's argument list: one is the pointer to the string, the other is the count.
`fixed`	a fixed-point number represented as an integer with 16 bits after the binary point

Table 5-1 *CPS Argument Types— Continued*

CPS type	*C type*
`token`	a special user-defined token used for performance improvement
`postscript`	`char *` sent to the server as POSTSCRIPT language code rather than as a POSTSCRIPT string
`cpostscript`	`char *` with an accompanying count sent to the server as POSTSCRIPT language code rather than as a POSTSCRIPT language string

Sending POSTSCRIPT Language Code without Returning Values

To create a `cdef` function that sends POSTSCRIPT language code to the server without requesting any results, you only need to use the following syntax, which omits the =>, *tag*, and *results* arguments:

`cdef` *name* `(`*args*`)` *POSTSCRIPT_code*

This type of `cdef` statement requires the name of the macro, the POSTSCRIPT language code, and possibly some arguments.

The following example shows how `cdef` was used to create the the standard CPS macro `ps_moveto()`.

```
cdef ps_moveto(x,y)  x y moveto
```

Thus, when the statement `ps_moveto(10,20)` is encountered in the `.c` file, the following POSTSCRIPT language code is transmitted:

```
10 20 moveto
```

Note that macros should always be structured to minimize the amount of traffic that occurs between client and server. For example, it may be useful to use `cdef` to define POSTSCRIPT language initialization routines that can themselves be called by subsequent `cdef` statements. This is shown by the following example:

```
cdef initialize()
        /draw-dot { 4 0 360 arc fill } def
cdef draw_dot(x,y)  x y draw-dot
```

Invoking `initialize()` transmits the definition of the POSTSCRIPT language function draw-dot a single time. Invocations of the routine `draw_dot()` from the C code — for example, `draw_dot(30,50)` — requires the transmission of fewer bytes than would be necessary if all the POSTSCRIPT language code were transmitted each time a dot were drawn.

Receiving Synchronous Replies

To create a `cdef` function that synchronously returns results from the server, the double symbol => must be used, followed by a *tag* and a *results* field, and some POSTSCRIPT code must be specified. Each argument in the *results* field must also have been specified in the *args* field; each argument will contain a value returned by the computation performed by the server. The *tag* argument will be specifed by the server as a small integer and associated with the *results*.

NeWS provides two operators that allow the server to return the specified *tag* and *results* arguments to the client program. The operators, which must be included in the `cdef` statement as part of the *POSTSCRIPT-code* argument, are as follows:

object **typedprint** –
Prints the object *object* in an encoded form on the current output stream. The object can then be read by the client program.

n **tagprint** –
Prints the integer *n* (where $-2^{15} \leq n < 2^{15}$) encoded as a tag on the current output stream.

Before calling the `cdef`, the client must define the tag to have some unique integer value. Note that the server does not force packets to begin with a tag and to contain typed data; this must be ensured by the client's POSTSCRIPT language code. The client should not pause in the middle of sending a tagged reply; if it does, the packet may be confused with a packet simultaneously returned as an asynchronous reply (see below).

The following is a generic syntax for requesting a synchronous tagged reply from the server:

```
#define tag tagint
cdef name(args) => tag (results) POSTSCRIPT-code
tag tagprint
typedprint
typedprint
```

In this syntax, *tag* represents the tag and *tagint* is the integer with which the tag is associated. Following the *POSTSCRIPT-code* used to perform the computation, **tagprint** is called with *tag* specified as its argument; thus, **tagprint** sends the defined value of *tag* (this value being *tagint*) back to the client. The **typedprint** operator is then called once for each argument specified in the *results* field; each result value is thus sent back to the client. (Note that **typedprint** can return variables of any of the CPS argument types.) The returned values can be then accessed within the C client according to their defined variable names, as specified in both the *args* and *results* fields.

Note that the *results* field is optional for a synchronous reply (and therefore the **typedprint** calls are also optional). Note also that the parentheses around the *results* are mandatory if the *results* field is used.

When a `cdef` function is used in this way, it transmits its *POSTSCRIPT-code* to the server, then pauses until the server sends the *tag* back with the accompanying *results*.

The following example demonstrates how to receive a synchronous reply by using a tagged *results* field.

```
#define BBOX_TAG 57
cdef ps_bbox(x0,y0,x1,y1) => BBOX_TAG (y1, x1, y0, x0)
    clippath pathbbox    % Find the bounding box of the
                         % current clip.
    BBOX_TAG tagprint    % Send back the tag.
    typedprint           % y1 is on the top of the stack,
    typedprint           % then x1. Thus, the results list
    typedprint           % is in the opposite order from
    typedprint           % the argument list.
```

This `cdef` statement defines a C function, named `ps_bbox()`, that takes as its parameters four pointers to integers. The function sets the integers to the bounding box of the current clipping path. When `ps_bbox()` is called, it transmits the specified POSTSCRIPT language code to the server. When executed, the `clippath pathbbox` call returns the bounding box of the current clipping region onto the operand stack; **tagprint** and **typedprint** then send back the tag and results to the C-client. The **tagprint** operator sends the tag 57 back to the client and the **typedprint** operators send back the coordinates. The C client has been waiting for the tag 57; thus, when the tag is returned, the client is able to receive the coordinates into the specified variables.

Receiving Asynchronous Replies

Typically, asynchronous replies are required when user input needs to be monitored. The client program enters a loop and, on each iteration, checks whether values have been returned from the server.

To create a `cdef` function that receives an asynchronous reply from the server, omit the *POSTSCRIPT-code* argument from the `cdef` statement:

`cdef` *name*`(`*args*`)` => *tag* `[(`*results*`)]`

When this form of `cdef` function is called from the C code, the input connection to the server is checked. The following then occurs:

- If no input is waiting, the client blocks until some input is sent from the server.
- If input is waiting (or arrives while the client is blocked), the first input item is compared with *tag*. If the input does not match the value of *tag*, the `cdef` returns 0 and the client continues execution. If the input does match the value of *tag*, the *results* are read into the specified variables and the `cdef` returns 1.

Thus, a `cdef` routine that sends no POSTSCRIPT language code to the server only blocks if no input has been sent from the server to the client; if input has been sent, execution of the client is allowed to continue even when the returned *tag* does not match.

The *results* field is optional for an asynchronous reply. The parentheses around the *results* are mandatory if the *results* field is used.

Note that the server is still responsible for executing **tagprint** and **typedprint** to return the specified *tag* and *results*. However, the code that calls these operators is not supplied by the `cdef` statement; instead, it must already have been sent to the server by a previous `cdef` statement that the client has executed. The code in the server is then triggered by an event (for example, the occurrence of user input) that is external to the client.

For example, the following expression, which sends a *tag* and *result* to a client, could be executed by the server whenever a menu selection is made by the user:

```
MENU_HIT_TAG tagprint
menuindex typedprint
```

Thus, the following `cdef` statement could be used within a client loop to receive asynchronous menu-selection messages:

```
cdef ps_menu_hit(index) => MENU_HIT_TAG (index)
```

When the function `ps_menu_hit()` is called from the client, the client blocks until input arrives from the server. When input arrives, the tag is compared to the `cdef` tag `MENU_HIT_TAG`. If the tag values match, `ps_menu_hit()` returns 1 and the value of the *results* field (in this case, an index) is received. If the tag values do not match, the function returns 0.

Functions such as `ps_menu_hit()` can be used to construct the basic command interpretation loops of a NeWS client program. This is demonstrated as follows:

```
while (!psio_error(PostScriptInput) {
        if (ps_menu_hit(&index))
                handle_menu_hit(index);
        else if (ps_character_typed(&character))
                handle_typed_character(character);
        else if (ps_redraw_requested())
                handle_redraw();
        else {
                /* illegal tag; program bug */
        }
}
```

5.3. Creating the .h File

To create a `.h` file, specify an existing `.cps` file as the argument to the `cps` command:

```
paper% cps test.cps
```

This creates a header file named `test.h`. The new file contains all definitions from the file `test.cps` in a specially compiled form. The `test.h` file can now be included in a `.c` file.

NOTE *For further information on the* `cps` *command and options, see the* `cps` *manual page.*

CPS Utilities

When you create a `.h` file with the `cps` program, additional utilities are automatically added to the `.h` file. You can use these utilities without defining them on the server side. The utilities, which are all found in `<NeWS/psmacros.h>`, are as follows:

Table 5-2 *C Utility Routines Provided by CPS*

Function()	*Description*
`ps_open_PostScript()`	open connection to X11/NeWS server
`ps_close_PostScript()`	close connection to X11/NeWS server
`ps_flush_PostScript()`	flush the output buffer
`ps_moveto(x,y)`	x y **moveto**
`ps_rmoveto(x,y)`	x y **rmoveto**
`ps_lineto(x,y)`	x y **lineto**
`ps_rlineto(x,y)`	x y **rlineto**
`ps_closepath()`	**closepath**
`ps_arc(x,y,r,a0,a1)`	x y r a0 a1 **arc**
`ps_stroke()`	**stroke**
`ps_fill()`	**fill**
`ps_show(string s)`	s **show**
`ps_cshow(cstring s)`	s **cshow**
`ps_findfont(string font)`	font **findfont**
`ps_scalefont(n)`	n **scalefont**
`ps_setfont()`	**setfont**
`ps_gsave()`	**gsave**
`ps_grestore()`	**grestore**
`ps_finddef(string font, usertoken)`	takes font, adds it to token list, and returns integer index of font into token list
`ps_scaledef(string font, scale, usertoken)`	takes font and scale, adds scaled font to token list, and returns integer index of font into token list
`ps_usetfont(token font)`	takes integer index of font into token list and sets current font to font given by token

5.4. Creating the `.c` File

The `.c` file, which is the main part of the C client program, should contain the following:

- An `#include` statement that references the `.h` file generated from the `.cps` file
- Calls to C macros originally defined in the `.cps` file

Note that variables used in the *results* arguments of the `cdef` command must be referenced as a pointer-variable combination in the argument list of the called macro.

- Standard CPS functions for managing communication between client and server

Three standard functions are used for opening communication, closing communication, and flushing data to the server. The functions are as follows:

- `ps_open_PostScript()`

  Establishes a connection to the X11/NeWS server specified by the `NEWS-SERVER` environment variable. If a connection to the X11/NeWS server is successfully established, `ps_open_PostScript()` returns a `PSFILE` pointer; if a connection is not established, it returns null. `ps_open_PostScript()` must be called before any procedure that needs to communicate with the server is called.

- `ps_flush_PostScript()`

  Output from the client to the server is buffered to ensure the efficiency of the interface mechanism: when the client calls a function that blocks while waiting for input, the contents of the buffer are automatically sent to the server. However, the client can send the contents of the buffer to the server at any time by calling `ps_flush_PostScript()`. The function returns -1 if an error occurs and 0 if no error occurs.

- `ps_close_PostScript()`

  Closes the connection to the server: this function should be called before the client program exits. The function returns -1 if an error occurs and 0 if no error occurs.

These functions form a subset of the CPS utilities automatically added to the `.h` file when the `cps` program is used. The utilities can be used in the C client provided that the CPS library is included when the client program is compiled. See the subsection *Compiling the* `.c` *File*, below.

POSTSCRIPT Language Communication Files

Two `PSFILE` pointers, `PostScript` and `PostScriptInput`, are the conduits through which information flows between the X11/NeWS server and the client program. When the client writes to the X11/NeWS server, it writes to the file represented by the pointer `PostScript`. When the client reads information sent from the server, it reads from the file represented by the pointer `PostScriptInput`. All operations on these `PSFILE` pointers are performed using the `psio` package, not the standard I/O package.

Reading the Client's Input Queue

This section describes the CPS library functions that examine the client's input queue. All the functions call I/O functions contained in the `psio` package. Note that the client can use these `psio` functions directly. However, if the client does so, it must explicitly pass the `PSFILE` structures for the files on which to read and write. CPS simplifies the task by supplying the `PSFILE` structures for you, using the pointers `PostScriptInput` and `PostScript`.

- `ps_check_PostScript_event()`

  Checks whether the input queue contains input. Returns 1 if the queue contains input, 0 if it does not, and -1 if there is an error.

- `ps_query_PostScript_event(tag)`

  Searches for *tag* in the input queue. Returns 1 if *tag* is in the queue, 0 if it is not in the queue, and -1 if there is an error.

- `ps_peek_PostScript_event(ptag)`

  Examines the tag associated with the top packet in the input queue and returns the tag's value in the pointer *ptag*. Th function leaves the tag in the queue. The function returns 1 if a tag is in the queue, 0 if something other than a tag is in the queue, and -1 if there is an error. If no input is in the queue when `ps_peek_PostScript_event` is called, the function blocks until the server sends input to the queue.

- `ps_read_PostScript_event(ptag)`

  This function is identical to `ps_peek_PostScript_event`, except that if a tag is found in the queue, `ps_read_PostScript_event` removes the tag from the queue. You should only use this function if you know that the tag in the queue has no associated data; otherwise, associated data is stranded in the input queue without a tag.

- `ps_skip_PostScript_event()`

  Removes the top entry from the input queue, regardless of what the entry is. Returns 1 if it successfully removes something from the queue and returns -1 if there is an error. If no input is in the queue when `ps_skip_PostScript_event` is called, the function blocks until the server sends input to the queue. This function can be used to remove a tag from the queue or it can be used to restore order in the input queue if a tag becomes separated from its associated data.

5.5. Tokens and Tokenization

NeWS provides a facility for establishing and maintaining a token list. This facility is intended for performance optimization. You should not use it unless your application is running and you are encountering problems with communication and interpretation overheads.

The token list is an efficient mechanism for the compression of data prior to transmission. The list is variable in length with a maximum dimension of 32768 elements. The first 32 elements are tightly compressed, yielding a 1-byte token. The next 1024 tokens generate two-byte codes. The remaining 31,712 generate three-byte codes.

Several operators are defined by the CPS utility to allow you to add and retrieve tokens from the token list. When a token is added to the list, it is available whenever the token is found by the scanner in the input stream. (It is frequently useful to add font objects to the token list and save the lookup time.)

NeWS has a mechanism, supported by CPS, where a client program and the server can cooperatively agree on the definition of a user token. The CPS declaration

```
usertoken black
```

tells CPS that you want to transmit the user-defined token `black` in compressed form. When `black` appears in following CPS definitions, the compressed token is used in the definition.

In order to establish the meaning of the token, the client has to talk to NeWS before the first use of the token. There are a number of procedures that the C program can call to do this:

- `ps_define_stack_token(`*utok*`)`

  Takes the value on the top of the stack in the server and defines it as the value of the token *utok*. In future messages to the server, *utok* has this value.

- `ps_define_value_token(`*utok*`)`

  Defines the user token *utok* to be the same as the current value of the POSTSCRIPT language variable *utok*. In future messages to the server, *utok* has the value that the POSTSCRIPT language variable *utok* had at the time `ps_define_value_token()` was called. Future changes to the value of the POSTSCRIPT language variable *utok*, or its identity as determined by changes in variable scope, have no effect on the definition of the token.

- `ps_define_word_token(`*utok*`)`

  Defines the user token *utok* to be the name of the POSTSCRIPT language variable *utok*. In future messages to the server, *utok* is the POSTSCRIPT variable *utok*. This binds the token *utok* to the name *utok*. When it is sent to the server, the name *utok* is evaluated and its value is used.

  The operators that manipulate the token list are listed in the table *C Utility Routines Provided by CPS.*

Compiling the `.c` File

When compiling the `.c` file, you must add the CPS library to the list of libraries searched by the linker. You must also inform the compiler and linker of the pathnames of these libraries and include files, using the `cc` options `-I` and `-L`. This can be achieved with a command line of the following form:

```
% cc -I$OPENWINHOME/include test.c -L$OPENWINHOME/lib -lcps
```

In this example, the pathnames provided to the compiler are the full pathnames of the CPS library and header files.

Comments

The CPS comment convention is the same as the POSTSCRIPT language comment convention: everything from a `%` sign to the end of a line is a comment. Despite the fact that CPS runs the `cpp` program on all input files, it uses the -C option so that the C-style commenting convention is interpreted as valid POSTSCRIPT language code.

5.6. Debugging CPS

You can test your application's POSTSCRIPT language code by typing into an interactive `psh` session with the server. However, you may reach a point at which the POSTSCRIPT language code only works in the context of the client-side program. Typically, a CPS program downloads a large amount of POSTSCRIPT language code and special operators in its "`initialize()`" `cdef` function. Therefore, you can place this portion of the POSTSCRIPT language code in a separate file and then change the initialization file to something that resembles the following:

```
cdef  initialize()
      (work/testinit.ps) LoadFile
      ... any other initialization required
```

You can now make changes to the POSTSCRIPT language initialization code (for example, adding "console (*debugging-statement* fprintf") in certain places) without having to recompile the C-side.

5.7. Supporting NeWS From Other Languages

The POSTSCRIPT language and C are the only languages that are supported for NeWS clients, the support for C being provided by the CPS preprocessor. For users who wish to download pure POSTSCRIPT programs to the server, equivalent mechanisms are provided by the `psh`(1) program.

If you wish to create a client process in some other language than NeWS or C, you must write a CPS-like program that is appropriate for the required language. The basis for such a program should be the input and output facilities used by CPS; the program should contain rountines for calling the facilities, macros that can be expanded into invocations of them, or similar features.

To provide runtime output for your CPS-equivalent program, you must create a function similar to the C function `pprintf()`, which provides the runtime output for the CPS program. This function is invoked in a manner similar to `fprintf()` (3S), taking a format string that is interpreted in a similar way. When the format strings contain `%s`, `%d`, or any of the other formatting specifiers, the corresponding arguments are transmitted as compressed binary tokens. The rest of the format string is transmitted as specified; it may contain compressed tokens or simple ASCII.

Input that the X11/NeWS server transmits to the client appears as bytes that can be read from the server I/O stream. The format of these bytes is specified entirely by the POSTSCRIPT language code downloaded by the client into the server; thus, it can be as simple or complex as is required. For interpreting these messages, NeWS contains a facility for writing objects back to the clients (using the same compressed binary format as the client uses to write to the server); CPS and a corresponding C procedure, named `pscanf()`. Several functions are also provided for detecting and manipulating tagged messages from the server. The tags are described below, in section 5.8, *Byte Stream Format.*

Contacting the Server

To contact the server from a UNIX environment you must obtain the correct IP port number of the server and connect to it. One way of obtaining the address is to examine the environment variable `NEWSSERVER`. This contains a string of the following form:

```
3227656822.2000;paper
```

The number before the period is the 32-bit IP address of the server in host byte order. The number after the period is the server's IP port number. To contact the server, you must create a socket and connect it to the IP address and port specified by these numbers. The name that follows the semicolon in the `NEWSSERVER` variable is the text name of the host on which the server is running; you can ignore this name.

The `newsserverstr(1)` command is a shell script that generates the appropriate string for `NEWSSERVER`.

Once a connection has been established, simply write bytes down the stream, as described below in section 5.8, *Byte Stream Format*. Remember, you do not need to use the compressed binary tokens, they are merely an optimization. ASCII POSTSCRIPT language code can be sent without use of compression.

5.8. Byte Stream Format

The information in this section is only of interest to programmers implementing the NeWS protocol. Most C programmers should use CPS, which deals with all of the protocol issues transparently.

The communication path between NeWS and a client is a byte stream that contains POSTSCRIPT programs. The basic encoding, which is compatible with POSTSCRIPT language printers, is simply a stream of ASCII characters. NeWS also supports a compressed binary encoding that may be freely intermixed with the ASCII encoding. The two encodings are differentiated according to the top bit of the eight-bit bytes in the stream. If the top bit is 0, the byte is an ASCII character. If it is 1, the byte is a compressed token. This differentiation is not applied within string constants or the parameter bytes of a compressed token.

Encoding For Compressed Tokens

Each compressed token has a code in its first byte; the code byte is a single byte with the top bit set. Parameter bytes may follow the code byte; parameters may also be encoded in the least significant bits of the code byte. The parameters are part of the token's description. After the code byte and any parameter bytes, there may be bytes that describe the token object itself, such as an encoded integer or string.

In the following description of the various types of token, the token values are referred to symbolically (with names). Some of the tokens use values taken from object tables; object tables are described below, in the subsection *Object Tables*.

enc_int

enc_int+(d<<2)+w ; $(w+1)*N$

$0 \le w \le 3$ and $0 \le d \le 3$: The next $w+1$ bytes form a signed integer taken from high order to low order. The bottom d bytes are after the binary point. This is used for encoding integers and fixed-point numbers.

enc_short_string	$\mathit{enc\_short\_string}+w$; $w*C$ $0 \le w \le 15$: The next w bytes are taken as a string.
enc_string	$\mathit{enc\_string}+w$; $(w+1)*L$; $l*C$ $0 \le w \le 3$: The next $w+1$ bytes form an unsigned integer taken from high order to low order. Call this value l. The next l bytes are taken as a string.
enc_syscommon	$\mathit{enc\_syscommon}+k$ $0 \le k \le 31$: Inside the X11/NeWS server there is table of POSTSCRIPT language objects. The **enc_syscommon** token causes the kth table entry to be inserted in the input stream. Typically these names are primitive POSTSCRIPT language operator objects. This table is a constant for all instances of POSTSCRIPT language code — the contents of the table are "well-known" and static. This token allows common POSTSCRIPT language operators to be encoded as a single byte.
enc_syscommon2	$\mathit{enc\_syscommon}\,2$; k $0 \le k \le 255$: This is essentially identical to **enc_syscommon** except that the index into the object table is $k+32$. This allows the less common NeWS operators to be encoded as two bytes.
enc_usercommon	$\mathit{enc\_usercommon}+k$ $0 \le k \le 31$: This is similar to **enc_syscommon** except that it provides user-definable tokens. Each communication channel to the server has an associated POSTSCRIPT language object table. The **enc_usercommon** token causes the kth table entry to be inserted in the input stream. The table is dynamic; it is the responsibility of the client program to load objects into this table. The NeWS operator **setfileinputtoken** associates an object with a table slot for an input channel.
enc_lusercommon	$\mathit{enc\_lusercommon}+j$; k $0 \le j \le 3$ and $0 \le k \le 255$: This is essentially identical to **enc_usercommon** except that the index is $(j<<8)+(k+32)$.
enc_eusercommon	$\mathit{enc\_eusercommon}$; jk $0 \le j \le 255$ and $0 \le k \le 255$: This is essentially identical to **enc_usercommon** except that the index is $(j<<8)+(k+32+1024)$.
enc_IEEEfloat	$\mathit{enc\_IEEEfloat}$; $4*F$ The next four bytes, high order to low order, form an IEEE format floating-point number.

enc_IEEEdouble

enc_IEEEdouble ; 8**F*

The next eight bytes, high order to low order, form an IEEE double precision floating-point number.

Object Tables

The **enc_*common*** tokens all interpolate values from object tables. The appearance of one of these tokens causes the appropriate object table entry to be used as the value of the token. These tokens are typically a part of a POSTSCRIPT language stream that is to be executed and can be any kind of POSTSCRIPT language object. Usually either executable keyword or operator objects are used.

This has some subtle implications with scope rules. If the object is a keyword, then its value will be looked up before being executed, just as an ASCII encoded keyword would be. If it is an operator object, then the operator will be executed directly, with no name lookup. This improves performance, but it also binds the interpretation of the object table slot at the time that the slot is loaded.

For example, if the executable keyword **moveto** were loaded into a slot, then whenever that token was encountered **moveto** would be looked up and executed. On the other hand, if the value of **moveto** were loaded into the slot, then whenever that token was encountered the interpretation of **moveto** *at the time the slot was loaded* would be used.

Magic Numbers

The binding between token names and values is as follows:

Table 5-3 *Token Values*

Octal	*Span*	*Symbolic Name*
0200	16	enc_int+(d<<2)+w
0220	16	enc_short_string+w
0240	4	enc_string
0244	1	enc_IEEEfloat
0245	1	enc_IEEEdouble
0246	1	enc_syscommon2
0247	4	enc_lusercommon
0253	1	enc_eusercommon
0254	4	free
0260	32	enc_syscommon
0320	32	enc_usercommon
0360	16	free

Examples

The POSTSCRIPT language code fragment

```
10 300 moveto
(Hello world) show
```

can be encoded simply as an ASCII text string

```
"10 300 moveto\n(Hello world) show "
```

that gives a message that is 33 bytes long. The space following **show** is a delimiter; without it the tokens would run together. Binary tokens are self-delimiting. If the tokens were sent in compressed binary format then the message would be the following 19 bytes:

Table 5-4 *Meaning of Bytes in Encoding Example*

Byte	*Meaning*
0200	Encoded integer, one byte long, no fractional bytes
0012	The number 10
0201	Encoded integer, two bytes long, no fractional bytes
0001	First byte of the number 300
0054	Second byte of the integer, (1<<8)+054==0454==300
0275	**moveto** is in slot 12 of the operator table
0233	(0220+11) Start of an 11-character string
0110	'H'
0145	'e'
. . .	
0144	'd'
0286	**show** is in slot 20 of the operator table

6

Debugging

6

Debugging

NeWS provides a *debugging facility* that allows you to set breakpoints and print messages to special output windows. The facility is provided as a POSTSCRIPT language extension file and can be modified by users.

This chapter describes the debugger and the operators it provides.

6.1. Loading the Debugger

The NeWS debugging facility is provided as the POSTSCRIPT language extension file `debug.ps`. Note that this file is not automatically loaded during the standard initialization process; thus, if you wish to load the debugger, you must execute the following command, either by adding it to your `.user.ps` file or by typing interactively in a **psh** window:

```
(NeWS/debug.ps) run
```

6.2. Starting the Debugger

To start the debugger, open a **psh executive** connection to the server and start the debugger with the **dbgstart** operator. This is demonstrated by the following example:

```
paper% psh
executive
Welcome to X11/NeWS Version 1.0
dbgstart
Debugger installed.
```

6.3. Using the Debugger

NeWS provides two kinds of debugging commands:

- Commands to be executed from client programs (*client commands*)
- Commands to be executed interactively by the user (*user commands*)

The **dbgstart** operator forks a debugger process that is attached to the `psh` connection and listens for debugger-related events generated by client commands. (All client commands broadcast debugger events to these debugger daemons.) Any client command that causes printing will print in each debugging `psh` connection.

Multi-Process Debugging

Since NeWS is a multi-process environment, you may often need to debug several processes at one time. The solution the debugger implements is to have each debugging connection maintain a list of processes that are paused for debugging. This list is printed via the **dbglistbreaks** command below. It is also printed whenever a new break occurs. Any of the listed breaks can be *entered* using the **dbgenterbreak** command. This swaps the `psh` debugging context out and replaces it with the paused process. The context currently consists of the dict stack and operand stack.

6.4. Client Commands

These are the client commands:

dbgbreak

name **dbgbreak** –

Causes the current client to pause, printing the pending breaks in all debugger connections. *Name* is used as a label in the list to distinguish between breaks, e.g. /Break1.

See also: **dbgbreakenter, dbgbreakexit**

dbgprintf

formatstring argarray **dbgprintf** –

Prints on each debugger connection, in **printf** style. If there are no debugger connections, it prints on the console. Thus the following code

```
(Testing: % % %\n) [1 2 3] dbgprintf
```

will print:

```
Testing: 1 2 3
```

on each debugger connection.

See also: **printf, dbgprintfenter , dbgprintfexit**

In addition to the above explicit calls to the debugger, errors cause the debugger to be implicitly invoked. This is done by the debugger putting a special error dictionary in the system dictionary. Each error slot in this debugger-supplied dictionary has a call to the debugger for each error. See the *PostScript Language Reference Manual* for details on error handling.

<errors>

– **<errors>** –
While debugging, a client error causes the client program to break to the debugger. This is *exactly* the same as inserting the code '/<errorname> dbgbreak' at the point the error occurred. Here is the result of encountering an **undefined** error while a debugger is running:

```
Break:/undefined from process(4154624, breakpoint)
Currently pending breakpoints are:
    1: /undefined called from process(4154624, breakpoint)
```

6.5. User Commands

Most of the user-level debugger commands come in two forms: one that explicitly takes a breaknumber and one that does not. The general rule is:

- A command of the form *cmdname***break** expects an explicit breaknumber for its argument.
- A command of the form *cmdname* (without "-break") uses an implicit breaknumber. This number is generally the currently entered break, or the last break in the list if there is no currently entered break.

The implicit form is primarily used in the most common case of only one break pending, or where constantly restating the breaknumber for the currently entered process would be arduous.

The user-commands are as follows:

dbgstart

– **dbgstart** –
Make the current connection to the server a debugger. Required before any of the other commands below can be used.

dbgstop

– **dbgstop** –
Removes the debugger from your `psh` connection.

dbglistbreaks

– **dbglistbreaks** –
List all the pending breakpoints resulting from **dbgbreak** and **<errors>** above. They are listed in the following form:

```
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
    3: /menubreak called from process(5177764, breakpoint)
    4: /undefined called from process(4154624, breakpoint)
```

The number preceding the colon is the *breaknumber* used in many of the following commands. A number beyond the end of the listing behaves as the last entry.

dbgbreakenter

name **dbgbreakenter** –
[dict name] **dbgbreakenter** –
Modify the named procedure to call **dbgbreak** just after starting. If the top of the operand stack is an array, it should contain a dict and the name of a procedure in the dict. Thus to break when any new window is made:

```
[DefaultWindow /new] dbgbreakenter
Break:/new from process(4050350, input_wait)
Currently pending breakpoints are:
    1: /new called from process(4050350, input_wait)
```

See also: **dbgbreak**

dbgbreakexit

name **dbgbreakexit** –
[dict name] **dbgbreakexit** –
Modify the named procedure to call **dbgbreak** just before exiting.
See also: **dbgbreak**

dbgremovebreak

breaknumber **dbgremovebreak** –
dbgremovebreak looks at the top execution stack entry of the stopped process. If it is a regular (non-packed) array and its current execution point is a call to **dbgbreak**, the **dbgbreak** is replaced with pop. Once a breakpoint is removed, the procedure must be redefined in order to put the breakpoint back.

dbgremove

– **dbgremove** –
dbgremove calls **debgmovebreak** on the currently entered breakpoint.

dbgprintfenter

name formatstring argarray **dbgprintfenter** –
[dict name] formatstring argarray **dbgprintfenter** –
Modify the named procedure to call **dbgprintf** with *formatstring* and *argarray* just after starting. Note that *argarray* can be an executable array if you want to defer evaluation of the arguments until the **dbgprintf** occurs.
See also: **dbgprintf**

dbgprintfexit

name formatstring argarray **dbgprintfexit** –
[dict name] formatstring argarray **dbgprintfexit** –
Modify the named procedure to call **dbgprintf** with *formatstring* and *argarray* just before exiting. The effects of this change will persist until the NeWS server is restarted. Note that *argarray* can be an executable array if you want to defer evaluation of the arguments until the **dbgprintf** occurs.

```
[DefaultWindow /reshape] (resize: % % % %\n)
   {FrameX FrameY FrameWidth FrameHeight} dbgprintfexit
resize: 91 100 179 181
resize: 91 94 223 187
```

See also: **dbgprintf**

dbgwherebreak

breaknumber **dbgwherebreak** –
Prints a exec stack trace for the process identified by *breaknumber*:

```
1 dbgwherebreak
Level 1
  { /foo 10 'def' /bar 20 'def' /A 'false' 'def' /B 'true'
    'def' /msg (Hi!) 'def' (Testing: %\n) 'mark' msg ] dbgprintf
    /oneB *dbgbreak } (*21,22)
Level 0
  { 100 'dict' 'begin' array{22} *'loop' 'end' } (*4,6)
```

The asterisk indicates the currently executing primitive in each level. The two numbers following each procedure are the index, relative to zero, of the asterisk and the size of the procedure. This is useful information for using **dbgpatch**.

dbgwhere

– **dbgwhere** –
Prints the execution stack for the currently entered process or for the last process listed if no process is currently entered.

dbgcontinuebreak

breaknumber **dbgcontinuebreak** –
Continues the process identified by *breaknumber*.

dbgcontinue

– **dbgcontinue** –
Continues the currently entered process or the last process listed if no process is currently entered.

dbgenterbreak

breaknumber **dbgenterbreak** –
As far as possible, make this debug connection have the same execution environment as the process identified by *breaknumber*. Currently, this includes the operand stack and the dictionary stack. Thus **dbgenterbreak** allows you to browse around in the given process' state. If **dbglistbreaks** is executed while within an entered process, the listing will indicate that process with a "=>" in the left margin:

```
3 dbgenterbreak
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
  =>3: /menubreak called from process(5177764, breakpoint)
```

dbgenter

– **dbgenter** –
Enters the last process listed.

dbgexit

– **dbgexit** –
Return to the debugger connection from whatever process you may have entered. This is a no-op if no process is currently entered. The following debugger primitives will call this routine: **dbgcontinuebreak, dbgkillbreak, dbgenterbreak, dbgstop.** Thus, **dbgenterbreak** first calls **dbgexit** to insure preserving state.

dbgcopystack

– **dbgcopystack** –
Copies the current operand stack to the process being debugged. This allows you to **dbgenter** a process, modify that copy of the operand stack, and copy it back to the process.

dbgcallbreak

arg clientproc breaknumber **dbgcallbreak** –
Execute **clientproc** in the broken process with *arg* as data. The **clientproc** primitive will be executed (in the client environment) with the *arg* on the stack, thus is responsible for popping it off.

dbgcall

arg clientproc **dbgcall** –
Implicit version of **dbgcallbreak**.

dbggetbreak

breaknumber **dbggetbreak** process
Returns the NeWS process object for the given breaknumber.

dbgpatchbreak

level index patch breaknumber **dbgpatchbreak** –
Patch the execution stack for breaknumber process. The patch overwrites the word in the executable at the given level, and at the given index within that level. Prints the resulting execution stack (**dbgwhere**).

dbgpatch

level index patch **dbgpatch** –
Patch the implicit process.

dbgmodifyproc

name/[dict name] headproc tailproc **dbgmodifyproc** –
Modify the named procedure to execute *headproc* just before calling it, and to call *tailproc* just after calling it. In affect, '{proc}' becomes '{headproc proc tailproc}.' This is the mechanism used for implementing **dbgbreakenter/exit** and **dbgprintfenter/exit.**

dbgkillbreak

breaknumber **dbgkillbreak** –
Kills a breakpointed process, removing it from the breaknumber list.

dbgkill

– **dbgkill** –
Kills the default process.

6.6. Miscellaneous Hints

Here are some miscellaneous tips for debugging.

Aliases

Because the debugger is based on the POSTSCRIPT language, the above commands can easily be modified or overridden entirely. One common change is to define some easily-typed aliases for the above verbose names. The following POSTSCRIPT language code does the trick; you can add this to your `.user.ps` file to make the aliases available in all debugging connections.

```
/dbe {dbgbreakenter} def
/dbx {dbgbreakexit} def
/dc  {dbgcontinue} def
/dcb {dbgcontinuebreak} def
/dcc {dbgcopystack dbgcontinue} def
/dcs {dbgcopystack} def
/de  {dbgenter} def
/deb {dbgenterbreak} def
/dgb {dbggetbreak} def
/dk  {dbgkill} def
/dkb {dbgkillbreak} def
/dlb {dbglistbreaks} def
/dmp {dbgmodifyproc} def
/dp  {dbgpatch} def
/dpe {dbgprintfenter} def
/dpx {dbgprintfexit} def
/dw  {dbgwhere} def
/dwb {dbgwherebreak} def
/dx  {dbgexit} def
```

Use Multiple Debugging Connections

If you are debugging POSTSCRIPT language code that you are running directly from an executive, start a debugging executive in another `psh` connection. This avoids having the debugging code trying to break to itself. You use the first executive to run the code being tested, and the second one to trap the errors.

7

Memory Management

7

Memory Management

In any software system, a limit must be imposed on the number of objects that are allowed to exist; otherwise, storage requirements become too great and performance is impaired. The usefulness of existing objects should thus be continually monitored; those that cease to be useful should be destroyed and their storage reclaimed.

NeWS provides a facility of *reference counting* that allows objects to survive as long as defined references to them exist; *references* are created by the system whenever one object becomes associated with another. When all references to an object are removed, the storage occupied by the object is automatically reclaimed.

This chapter explains the principles of reference counting; it also lists the operators that NeWS provides for the purposes of memory management.

7.1. Reference Counting

NeWS counts references that are made to a given object, using this to determine how long the object is maintained in storage. The operations that are applied to an object have the effect of adding or removing references to that object.

Objects

For memory management purposes, two kinds of object exist:

- Uncounted objects

  These are simple resources, such as booleans, fixed numbers, and real numbers. These objects are not shared and therefore have no reference count.

- Counted objects

  These objects, which include all other resources that the system contains, can be shared within the system. Thus, they are reference counted and can be systematically removed when they become useless.

Objects are grouped according to type. A complete list of object types is provided below, in the subsection *Object Types*.

References to Counted Objects

A *counted object* is an object that may have one or more references indicating its existence; a *reference* is a NeWS pointer that extends from one object to another. NeWS supports two kinds of reference, *counted* and *uncounted.*

Counted References

A *counted reference* affects the existence of an object. While the object has at least one counted reference, the object continues to exist in memory, and its storage cannot be reclaimed.

Note that a reference is always considered to be the property of its recipient; thus, if a reference points from A to B, the reference belongs to B and is included in B's reference count; moreover, the existence of the reference ensures that B remains in storage. By contrast, when A is destroyed, its reference to B is destroyed and B becomes available for garbage collection, provided that no other counted references point to it.

See the subsection *Reference Tallies* below for information on how references are counted.

Uncounted References

An *uncounted reference*, which is created only by the server itself, never affects the existence of a object; it is ignored by the reference counting and automatically cleaned up by the garbage collection procedures.

Uncounted references are used to avoid *circular references*; these occur when two objects point to each other with counted references; neither object can be destroyed, since each continues to be referenced by the other. To prevent this from occurring in the canvas hierarchy, NeWS ensures that a parent canvas always references its child canvas with an *uncounted reference*; this allows the child canvas to be destroyed and its storage immediately reclaimed, provided that no other counted references to the child exist. However, a child canvas always references its parent with a *counted reference*; thus, the parent is never destroyed while any of its children exist, regardless of the removal of other references to the parent.

NOTE *The uncounted reference that points from a parent canvas to its child is used by NeWS internals to perform access operations such as the following:*

```
NewCanvas /TopChild get
```

When this code is entered, NeWS locates the top child of the specified canvas by tracing the appropriate uncounted reference.

Soft References

Since uncounted references are created only by the server, NeWS programmers cannot use them to prevent circular references from occurring in the objects they themselves define; instead, programmers must use *soft references.* A soft reference is created with the following operator:

any **soften** any

The operator takes a single argument and returns it unchanged, except that if the argument is a reference to an object, it is returned as a soft reference. If the operator is used to soften the last existing hard reference to an object, the object becomes obsolete and an *obsolescence event* is generated by the system.

NOTE *Objects in NeWS are frequently pointed to by many references; thus, when using the* **harden**, **soft**, *and* **soften** *operators, the programmer is responsible for specifying the correct reference to be operated upon.*

A *soft* reference differs from an *uncounted* reference, since it exists as a *counted reference* and ensures the continued existence of the object to which it points. However, to allow soft references to govern storage reclamation, NeWS associates them with *obsolescence events*.

Obsolescence Events

An *obsolescence event* is automatically generated by the system when an object is preserved only by soft references (that is, when all the remaining references to it are soft). The event, which has **Obsolete** in its **Name** field and a copy of the object in its **Action** field, signifies that all remaining references to the object are soft.

Any process that uses the **soften** operator to soften an existing hard reference should also express interest in receiving an *obsolescence* event: when the event is distributed and successfully matched to the interest, the event can be passed to a handler that removes the process' soft reference. When all references have been removed, the object is automatically garbage collected.

For further information on event management, see Chapter 3, *Events*.

NOTE *Soft references can also used by any NeWS system process that tracks resources within the system; an example of such a process is a window manager, which tracks windows. When all other references to the window are removed, the window manager can respond to the consequent system-generated obsolescence event by removing its own soft reference. This prevents a useless window from continuing to exist due to its link with the window manager.*

Reference Tallies

Each counted object contains two tallies, which are as follows:

- The total number of counted references to the object
- The number of those references to the object that are soft

Object Types

This section lists all existing object types. The types are presented in two tables: the first table contains uncounted objects; the second, counted objects.

Table 7-1 *Uncounted Object Types*

booleantype	**marktype**	**operatortype**
colortype	**nametype**	**realtype**
integertype	**nulltype**	**savetype**

Table 7-2 *Counted Object Types*

arraytype	**eventtype**	**processtype**
canvastype	**filetype**	**stringtype**
colormapentrytype	**fonttype**	**visualtype**
colormaptype	**graphicsstatetype**	
cursortype	**monitortype**	
dicttype	**packedarraytype**	
environmenttype	**pathtype**	

7.2. Memory Management Operators

This section lists the operators that are provided by NeWS for the purposes of memory management.

harden

any **harden** any
Takes a single argument and returns it unchanged, except that if the argument is a soft reference to an object, a hard reference to the same object is returned.

Caution should be exercised when using this operator; results may not be as expected depending on the state of the target object. For example, suppose the target is an obsolete canvas on which most obsolescence handling has been performed. If some of the soft references that have been removed belonged to system processes such as the window manager, hardening a remaining soft reference will keep the canvas in existence, but it will not be tracked in the system as it had been.

soft

any **soft** boolean
Takes a single argument and returns `true` if the argument is a soft reference to an object, `false` otherwise.

soften

any **soften** any
The operator takes a single argument and returns it unchanged, except that if the argument is a reference to an object, it is returned as a soft reference. If the operator is used to soften the last existing hard reference to an object, the object becomes obsolete and an *obsolescence event* is generated by the system.

NOTE *Objects in NeWS are frequently pointed to by many references; thus, when using the* **harden**, **soft**, *and* **soften** *operators, the programmer is responsible for specifying the correct reference to be operated upon. This is demonstrated by the following example:*

```
/cv framebuffer newcanvas def        % This creates a single hard
                                     % reference in the process'
                                     % dictionary.

cv setcanvas                         % This creates a second hard
                                     % reference, extending from
                                     % the graphicsstate to the
                                     % current canvas.

/cv cv soften def                    % This softens the cv reference.

currentcanvas soften setcanvas       % This softens the graphicsstate
                                     % reference.
```

7.3. Memory Management Debugging Operators

This section lists the operators that are provided by NeWS for the purposes of memory management debugging.

Using the debugdict

With the exception of the operator **vmstatus**, the debugging operators described in this section are contained in a NeWS system dictionary named **debugdict**. Therefore, before typing any of these debugging operators in the **psh** connection, you must type **debugdict begin** to place **debugdict** on the dictionary stack. When you type **end**, the dictionary is closed. If you attempt to use the debugging operators while the dictionary is closed, the system signals `undefined` errors.

Thus, you must use the debugging operators as follows:

```
debugdict begin
 debugging operators
  .
  .
end
```

Debugging Operators

The debugging operators are as follows:

objectdump

file **objectdump** –
Writes to the specified file a formatted summary of the number of objects that the server has created. Note that the specified *file* must be open for writing; otherwise, an `invalidaccess` error is signaled.

In the output, objects are classified according to the following *families*:

- *interpreter data*

  Contains objects allocated by the interpreter for the execution of POSTSCRIPT language code.

- *core types*

  Contains objects allocated for NeWS language processes; any of these objects can appear on the operand stack of a process.

- *shapes classes*

  Contains objects allocated by the underlying graphics library to perform rendering.

- *miscellaneous types*

  Contains objects allocated for system management, such as the overhead incurred processing fonts and the memory allocated to support the unused font cache (see the following section for a discussion about this).

- *other*

  Contains I/O buffer space, objects for which accounting is not performed, and memory allocator overhead.

The output written to the specified file has the following form:

```
family_name family:
    nnnnn bytes for mm object_type objects
```

The operator is demonstrated by the following **psh** example:

```
debugdict begin                          % Use the debugging dictionary.
currentfile objectdump                   % This directs output to the
                                         % psh connection.

(/tmp/objects1) (w) file objectdump      % This directs output to the
                                         % specified file.
/new MyClass send
(/tmp/objects2) (w) file objectdump      % A second file is specified; the
                                         % two files can now be compared
                                         % to indicate the number of
                                         % objects created due to the
                                         % creation of an instance
                                         % of MyClass.

end                                      % End use of the debugging
                                         % dictionary.
```

refcnt

object **refcnt** fixed fixed

Returns two numbers onto the process stack: the first is the total reference count for the specified object; the second is the soft reference count for the specified object. (Note that the counts indicate the status of the object after the operator has cleaned up the reference to the object that was given to it on the stack.)

reffinder

object **reffinder** –
object boolean **reffinder** –

Prints to standard output information on all current references to the specified object. The optional *boolean* argument can be `true` (which specifies that only information about references that are not soft is printed) or `false` (which specifies that information about all references is printed).

If the specified object is not a counted type, a message is printed and **reffinder** returns.

The **reffinder** operator causes memory to be allocated for a hash table, which holds traceback information about the system. All allocated memory is freed when the operator returns. If memory cannot be allocated, a message is printed, all memory currently allocated due to the operator is freed, and the operator returns.

The **refcnt** and **reffinder** operators can be used together; **refcnt** determining how many references to an object continue to exist, **reffinder** printing information on those references. Note that if a call to **refcnt** indicates that all but one of an object's remaining references are soft, the problem you are debugging is likely to have been caused by the remaining non-soft reference. In such a case, **reffinder** should be executed with the *boolean* argument specified as `true`: this prints information on the non-soft reference only.

Use of **reffinder** may indicate a discrepancy between the number of references registered in the object's tallies and the number that actually exist in the system. If this occurs, messages are printed to indicate the discrepancy. There are at least two possible reasons why the discrepancy might occur:

- A *cycle* exists; that is, an object in the system contains a reference to itself. Note that the **reffinder** operator cannot find cycles.

  The effects of a cycle can be illustrated as follows. The **reffinder** operator is used to search for references to a canvas. A reference to the canvas is held by a dictionary. The dictionary holds a reference to itself; however, no external references to the dictionary exist. Therefore, **reffinder** cannot find the dictionary, since there are no external references to it. Since it cannot find the dictionary, it cannot find the dictionary's reference to the canvas either.

- A reference counting bug exists in the server. While possible, the likelihood of this happening is small. User code should be examined thoroughly for cycles and errors in cleanup processing.

If such a discrepancy is reported, proceed as follows:

- If all existing references to the object are soft, check obsolete processing to be sure it is being invoked correctly and that, once invoked, it is executing correctly.
- Check code (particularly POSTSCRIPT language code) for reference counting bugs and cycles.

vmstatus

- **vmstatus** num num num

Returns three numbers, which indicate the amount of available memory, the amount of memory used, and the system break value.

NOTE *The information returned by this NeWS operator differs from that returned by the standard POSTSCRIPT language* **vmstatus** *operator.*

The **vmstatus** *operator is provided as a standard NeWS operator and is not part of* **debugdict**, *as are the other operators described in this section.*

7.4. The Unused Font Cache

In NeWS, the memory requirements for font representation may be high, particularly when an application uses multiple fonts or employs a wide range of font-sizes. It is important, therefore, that no font should ever occupy memory unnecessarily.

However, it is often inappropriate to remove a font from memory when its reference count becomes zero: the font may need to be used again, and the performance cost of reloading fonts is high. Therefore, to prevent the unnecessary reloading of fonts, NeWS provides an *unused font cache.* When a font's reference count becomes zero, it is not destroyed; instead, it is placed in the unused font cache. The font thus continues to exist in memory and is not destroyed. When the font is subsequently referenced, it is removed from the cache.

The size of the cache is limited at all times; the limit can be determined by the user (see the following subsection for details). If the cache becomes full, and a new font needs to be added, the earliest cached font is removed and destroyed; its memory is thus freed. Fonts continue to be removed from the cache and deleted from memory until sufficient room for the new font has been created. (Note that fonts may be of different sizes and may thus maintain different memory requirements: therefore, no precise figure exists for the number of fonts that may be cached at one time.)

The size limit of the unused font cache determines the balance between memory consumption (caused by maintaining fonts in memory) and performance degradation (caused by reloading fonts). When the size limit is high, fonts tend to be maintained in memory; when the limit is low, fonts tend to be destroyed.

Specifying the Size of the Cache

Memory is not allocated for the unused font cache. The size of the cache is the amount of space in the system consumed by unused fonts; it is set to a default value during system initialization. The following operators can be used to query and set the size of the cache:

currentfontmem

– **currentfontmem** num

Returns the size of the font memory cache in units of kilobytes: this is the amount of memory that is used to store unused fonts in the system.

setfontmem

size **setfontmem** –

Sets the size of the font memory cache. The *num* argument specifies the size of the cache in units of kilobytes. This is the amount of memory that is used to store unused fonts in the system.

If a font that is bigger than the current size limit of the cache is itself cached, the cache is automatically expanded by the size of the new font. After this has occurred, the size of the cache can only be decreased by use of the **setfontmem** operator.

For most font uses, the default cache size is sufficient. However, if you run applications that use multiple fonts, and some of the fonts are large, an expanded cache may be required to avoid the appearance of performance degradation.

Flushing the cache

To flush the cache, execute **setfontmem** with *size* set to 0. This frees the memory of all unused fonts. If the system is run with the cache size set to zero, the memory of each font is freed whenever its reference counts go to zero. Running the system with a cache size of zero is not recommended, due to the performance penalties associated with loading fonts.

Applications

Applications cannot ascertain current system memory: thus, they should never attempt to modify the cache size. Setting an improperly high cache size may consume all available memory and cause the server to crash.

8

NeWS Type Extensions

8
NeWS Type Extensions

NeWS extends the POSTSCRIPT language with a number of new types. These new types are necessary because NeWS programs run in a dynamic, interactive environment whereas traditional POSTSCRIPT programs run inside a printer. The type extensions allow NeWS to support multiple imaging surfaces, user input, multiple processes, and the other requirements of a window system.

In addition to the type extensions, NeWS defines a number of operator extensions to support the new types. The operator extensions are described in the next chapter.

Some of the NeWS type extensions are opaque and can only be used with operators that have been created or extended to handle them. Other types behave just like dictionaries, and all dictionary access operators can be used on them. This chapter describes all the NeWS type extensions.

8.1. NeWS Objects as Dictionaries

Some NeWS type extensions have pieces of internal state that are accessible to the NeWS programmer. Objects of these types behave almost exactly like standard POSTSCRIPT language dictionary objects. All of the standard dictionary manipulation operators (such as **begin**, **def**, **get**, and **put**) work on these new types. However, the internal representation of these objects is completely different from standard dictionaries, and storing or retrieving values from these new types may involve side-effects. Objects of these new types are known as *magic dictionary* objects.

Although magic dictionaries are extremely similar to standard PostScript language dictionaries, several important differences exist:

- Magic dictionary objects are not created with the **dict** operator as are standard dictionaries. Instead, magic dictionary objects are created with special operators. NeWS provides one creation operator for each magic dictionary type. For example, the **newcanvas** operator creates a new canvas, and the **createevent** operator creates a new event.
- Magic dictionary objects contain predefined key-value pairs, which cannot be removed with the **undef** operator. The key in each pair names a piece of internal state of the object, and the value is a POSTSCRIPT language representation of that state. These predefined keys are already present in newly-created magic dictionary objects.

- Some of the predefined key-value pairs are read-only. Attempts to write these key-value pairs (for example, with **put** or **def**) will result in an `invalidaccess` error.

The following examples illustrate the use of operator extensions and standard dictionary operators to manipulate magic dictionary objects. The examples use **canvastype** and **eventtype** objects. These types are explained in greater detail later in this chapter.

```
/MyCanvas framebuffer newcanvas def  % Create a new canvas as a child of
                                     % the framebuffer and store it in
                                     % /MyCanvas in the current
                                     % dictionary.

MyCanvas /Mapped true put            % Set the mapped state of MyCanvas
                                     % to be true. This has the side
                                     % effect of painting the contents
                                     % of the canvas to the screen.

MyCanvas /Color get                  % Retrieve the Color attribute of the
                                     % canvas, a boolean value. This
                                     % attribute is read-only and cannot
                                     % be changed.
```

8.2. List of NeWS Types

This section lists all the types that are accessible to NeWS programmers.

POSTSCRIPT Language Types

NeWS provides the following standard POSTSCRIPT language object types, which are returned by the **type** operator:

Table 8-1 *Standard Object Types in the POSTSCRIPT Language*

arraytype	**marktype**	**realtype**
booleantype	**nametype**	**savetype**
dicttype	**nulltype**	**stringtype**
filetype	**operatortype**	
integertype	**packedarraytype**	

All of the above types, except **packedarraytype**, are described in the *PostScript Language Reference Manual*. The **packedarraytype** is a new POSTSCRIPT type that will be included in a future edition of the *PostScript Language Reference Manual*.

NeWS Type Extensions

The following types are provided by NeWS as extensions to the POSTSCRIPT language:

Table 8-2 *Additional NeWS Object Types*

canvastype	**environmenttype**	**pathtype**
colormapentrytype	**eventtype**	**processtype**
colormaptype	**fonttype**	**visualtype**
colortype	**graphicsstatetype**	
cursortype	**monitortype**	

All of the above types are accessible as POSTSCRIPT language dictionaries except for **colortype**, **graphicsstatetype**, **monitortype**, and **pathtype**.

The **type** operator returns the name of all of the above NeWS type extensions except for **fonttype**; the **type** operator returns **dicttype** for a NeWS font object to be consistent with the POSTSCRIPT language. The NeWS **truetype** operator returns **fonttype** for a NeWS font object.

The sections that follow provide a description of each NeWS type extension. For each type that is accessible as a dictionary, the dictionary keys are described. The types that are not accessible as dictionaries are listed first; the other types are listed in alphabetical order. Because **packedarraytype** is not yet described in the *PostScript Language Reference Manual*, it is described here along with the NeWS type extensions.

8.3. colortype

NeWS *color* objects can have either *red/green/blue* or *hue/saturation/brightness* values. Color objects with red/green/blue components are created with the **rgbcolor** operator. Color objects with hue/saturation/brightness components are created with the **hsbcolor** operator. The color objects can be compared and can be used as a source of paint for the rendering primitives. Color objects cannot be accessed as dictionaries.

NOTE *NeWS provides a dictionary of named colors; see Chapter 10, Extensibility through POSTSCRIPT Language Files for information.*

8.4. graphicsstatetype

Graphics state objects preserve entire graphics states, as defined by the POSTSCRIPT language, in a permanent form. Their only use is to save the graphics state of a process for future re-use by that (or another) process. They are retrieved and set with the **currentstate** and **setstate** operators. They cannot be accessed as dictionaries.

8.5. monitortype

Monitor objects can be accessed by only one process at a time; they are used for synchronization. A monitor object can be locked or unlocked. Processes can use monitors to implement mutual exclusion (for example, to prevent conflicts in updating shared data structures). Monitors are created with the **createmonitor** operator. Monitors cannot be accessed as dictionaries.

8.6. packedarraytype

The NeWS **packedarraytype** is equivalent to the new POSTSCRIPT **packedarraytype**; **packedarraytype** is documented here because it is not yet included in the *PostScript Language Reference Manual.*

A **packedarraytype** object is a more compact representation of an array than an ordinary **arraytype** object. *Packed arrays* save space; they should be used whenever possible.

In many ways, packed array objects are similar to ordinary array objects. Packed arrays can be executed. Elements or subarrays can be extracted from a packed array with the standard **get** and **getinterval** operators; a subarray of a packed array is itself a packed array. Packed arrays can be enumerated with the **forall** operator.

However, some differences do exist between packed array objects and ordinary array objects. Packed arrays are always read-only; the **put** operator cannot be used to store into a packed array. Accessing arbitrary elements of a packed array can be slow, but accessing the elements sequentially takes about the same amount of time as it does for an ordinary array.

The **setpacking** operator can be used to set a process' array-packing mode to true; when true, the server automatically creates packed arrays for any executable array that it reads for that process. When the symbol "{" is encountered, the server accumulates all tokens until the associated "}" and then creates a packed array instead of an ordinary array. The array-packing mode defaults to false. A child process inherits its parent's array-packing mode.

A packed array can also be created with the **packedarray** operator. This operator takes as arguments the objects that are to be included in the packed array.

8.7. pathtype

Path objects represent paths, as defined by the POSTSCRIPT language, in a permanent form. Their only use is to save the current path of a process for future re-use by that (or another) process. They are retrieved and set with the **currentpath** and **setpath** operators. They cannot be accessed as dictionaries.

8.8. canvastype

All NeWS *canvas* objects are of type **canvastype**. Each canvas is a surface on which objects such as text or graphic images can be drawn. A canvas' boundary is represented by a POSTSCRIPT language path and can be any arbitrary shape. When mapped to the screen, canvases can overlap. When this occurs, the hidden portion of a canvas can be stored offscreen and redisplayed when the canvas is re-exposed.

Canvases exist in a hierarchy. The background of the screen is the root of the hierarchy and is thus known as the *root canvas*. A canvas can have any number of children, each of which can exist at any coordinates; however, each child is visually clipped by the bounds of its parent and thus becomes invisible when located outside those bounds.

Canvases are created with the **newcanvas** operator. They can be accessed as dictionaries.

Canvases are described in detail in Chapter 2, *Canvases*. This section describes the keys in the canvas dictionary.

A **canvastype** dictionary contains the following keys:

TopCanvas
BottomCanvas
CanvasAbove
CanvasBelow
TopChild
Parent
Transparent
Mapped
Retained
SaveBehind
Color
EventsConsumed
Interests
Cursor
Colormap
Visual
VisualList
OverrideRedirect
BorderWidth
UserProps
XID
SharedFile
RowBytes
Grabbed
GrabToken

The value of each key is described below.

TopCanvas canvas (*read-only*)
The canvas' top sibling (the **TopChild** of the parent canvas), or the canvas itself if it has no siblings.

BottomCanvas canvas (*read-only*)
The canvas' bottom sibling (the bottom child of the parent canvas), or the canvas itself if it has no siblings.

CanvasAbove canvas *or* null
The sibling canvas immediately above this canvas, or null if no such canvas exists. You can change a canvas' position in the hierarchy by setting the value of this key to be any of the canvas' siblings. When you set the value of a canvas' **CanvasAbove** key, the canvas is inserted into the hierarchy directly below the specified sibling. Note that the **CanvasAbove** and **CanvasBelow** keys of the affected siblings will change to reflect the new hierarchy.

CanvasBelow canvas *or* null
The sibling canvas immediately below this canvas, or null if no such canvas exists. You can change a canvas' position in the hierarchy by setting the value of this key to be any of the canvas' siblings. When you set the value of a canvas' **CanvasBelow** key, the canvas is inserted into the hierarchy directly above the specified sibling. Note that the **CanvasAbove** and **CanvasBelow** keys of the affected siblings will change to reflect the new hierarchy.

TopChild canvas *or* null (*read-only*)
The top child of this canvas, or null if no such canvas exists.

Parent canvas *or* null
The parent of this canvas, or null if the canvas has no parent. Null is associated with canvases that result from **createdevice**, **readcanvas**, and **buildimage**. Setting a canvas' **Parent** key manipulates the canvas hierarchy; the canvas becomes the top child of the canvas specified in this key. Canvases created with **readcanvas** and **buildimage** cannot be inserted into the canvas hierarchy; setting the **Parent** key of such a canvas is ignored.

Transparent boolean
True if the canvas is transparent, false if it is opaque. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. An opaque canvas can be damaged; a transparent canvas cannot. A transparent canvas never has a retained image; instead it shares its parent's retained image. Anything painted on a transparent canvas is actually painted on the first opaque canvas beneath it (often, its parent).

Mapped boolean
True if the canvas is mapped, false if it is unmapped. When a canvas is mapped, it becomes visible on the screen that its parent is on, provided that all of its ancestors are mapped and that it is not obscured by overlapping canvases. Note that canvases created with **readcanvas** and **buildimage** cannot be mapped to the screen. When a nonretained canvas is mapped, the region that becomes visible is considered to be damaged.

Retained boolean
True if the canvas is retained, false if it is not. NeWS keeps an offscreen copy of the invisible parts of a retained canvas. If a retained canvas is mapped and is overlapped by some other canvas, the hidden parts of the canvas will be saved. If a canvas is retained when it is not mapped, a copy of the entire canvas is saved.

A retained canvas usually performs much better with most window management operations, like moving and mapping canvases. But the retained image does consume storage. For color displays, the cost of retaining canvases is often prohibitive.

If the server runs low on memory, the retained portions of canvases may be reclaimed. When this happens, querying the **Retained** field of such a canvas returns false. In addition, damage may be reported on this canvas. Therefore, programs should be prepared to handle damage on any canvas, including retained ones.

The **Retained** field is meaningless for a transparent canvas. When queried, it returns the **Retained** value of its nearest opaque ancestor; in this case, the value cannot be changed.

SaveBehind boolean
SaveBehind is a hint to the window system that when the canvas is made visible on the screen, the canvas won't be up very long and the canvases below it won't be very active. If the value of a canvas' **SaveBehind** key is true, NeWS usually saves the values of the pixels underneath the canvas when the canvas is mapped to the screen. NeWS then restores the original pixel values back to the screen when the canvas is unmapped, and none of the canvases are damaged. This is a performance hint only; it does not affect the semantics of any other operations. It is usually employed with pop-up canvases to reduce the cost of damage repair when they are unmapped.

Color boolean (*read-only*)
True if and only if this canvas can support more colors than just black-and-white or greyscale.

EventsConsumed name
This key determines the event consumption behavior of the canvas. Its value is one of the following names:

/AllEvents
All events that are tested against this canvas' post-child interests are consumed; they are not tested against the post-child interest lists of this canvas' ancestors.

/MatchedEvents
Events that match a post-child interest of this canvas are consumed, but non-matching events may still pass to this canvas' ancestors for further testing against post-child interests.

/NoEvents
No events are consumed by this canvas; all events may pass to the canvas' ancestors during testing against post-child interests.

Interests array (*read-only*)
The interest lists for the canvas, represented as an array of events. The array is a concatenation of the canvas' pre-child and post-child interest lists, with the pre-child interest list first. Within each list, the interests are ordered according to their priority, with highest priority first. Among interests with the same priority, exclusive interests are listed first.

Cursor cursor *or* null
The cursor associated with this canvas, or null if a cursor has not been specified for this canvas.

Colormap colormap
The colormap that is associated with this canvas (see **colormaptype**).

Visual visual (*read-only*)
The visual that is associated with this canvas (see **visualtype**).

VisualList array (*read-only*)
An array that contains all possible visuals for the canvas (see **visualtype**).

OverrideRedirect boolean (*read-only*)
True if an X11 client has selected the **OverrideRedirect** window attribute for this canvas. (This key is useful only for canvases created by X11.)

BorderWidth null *or* integer (*read-only*)
The X11 border width. If this value is an integer, the canvas has a window border with the specified width. A non-null **BorderWidth** can be changed with the **reshapecanvas** operator. If the value is null, the canvas has no border and none can be set. (This key is useful only for canvases created by X11.)

UserProps dict
A dictionary that contains the X11 properties for this canvas. The keys in this dictionary are atoms that name each property. The values are arrays of length four, containing the property name, type, format, and data in that order. (This key is useful only for canvases created by X11.)

XID number (*read-only*)
The X11 resource ID of the canvas. If this number is zero, the canvas is not in the X11 resource database. (This key is useful only for canvases created by X11.)

SharedFile string
Maps a canvas object to a file. The canvas must have been previously defined with the **buildimage** operator. Note that canvases defined with **buildimage** have no parent. If the specified canvas does have a parent, a `typecheck` error is returned. The *string* must contain the name of a file in the server's name space. If the file is inaccessible or does not have read-write access permission, an `invalidfileaccess` error is returned. If the canvas is currently mapped to a file, a null *string* unmaps the file; a *string* containing a filename unmaps the current file and remaps the canvas to the file named in *string*. The file is assumed to contain image data stored a line at a time in increasing y order, the number of bytes per scanline being that specified by **RowBytes**.

The ability to map a canvas to a file is operating system dependent and may not be present in the server.

The file should be accessed by the client directly using `mmap(2)`. The server will process the shared file in native byte order. The client is responsible for synchronizing accesses to the shared file. This facility is intended for use by clients running locally with the server. If the client and server do not reside on the same machine, canvas data consistency is not guaranteed by the server.

RowBytes number (*read-only*)
The scanline padding requirements for a canvas. This represents the dimensioned width plus any padding added by the server.

Grabbed boolean
Unless you are using a GX graphics accelerator, neither this key nor the **GrabToken** key (see below) has an effect on the canvas.

If you are running X11/NeWS with a GX graphics accelerator (`FRAMEBUFFER=/dev/cgsix0`), this key controls NeWS access to the graphics hardware. When used in conjunction with a C language interface to the hardware, the key mediates the control over the bits inside a given canvas. To demonstrate how the **Grabbed** key is used, the following code creates a new canvas:

```
/can framebuffer newcanvas def
```

The following example shows the three possible uses of the **Grabbed** key:

```
can /Grabbed true put          % Make can a grabbed window.
can /Grabbed false put         % Release the grab on can.
can /Grabbed get               % Returns the value of
                               % the Grabbed key.
```

When a GX graphics accelerator is present and a client sets a canvas' **Grabbed** key to true, the cgsix segment driver assigns an integer to the canvas' **GrabToken** key. The client can then communicate with the cgsix segment driver using this **GrabToken** to identify which canvas' clip area to use when rendering directly to the framebuffer.

GrabToken int (*read-only*)
The grab token for the canvas. This key's value is zero when the canvas is not grabbed and is a non-zero integer when it is grabbed. The key is demonstrated by the following example:

```
can /GrabToken get             % Returns 0 if not grabbed.
```

8.9. colormaptype

A *colormap* is a color lookup table that determines which color is displayed for a specified pixel value. Each entry in the colormap table contains a red, a blue, and a green value; these values can be used to specify the color-mix of a given pixel. Each entry also contains an integer that is a index for the entry.

A colormap can be created with the **createcolormap** operator (see Chapter 9, *NeWS Operator Extensions*). Colormaps can be accessed as dictionaries.

A **colormaptype** dictionary contains the following keys:

Entries
Free
Installed
Visual

The value of each key is described below.

Entries array (*read-only*)
An array of the colormapentries used by this colormap. The mininum number of elements in the array is 0; the maximum number of elements is given by the **Size** key of the colormap's visual.

Free number (*read-only*)
The number of free entries in the colormap.

Installed boolean
True if the colormap is installed as a hardware map; otherwise false.

Visual object (*read-only*)
An object that is the visual for this colormap. Note that a canvas and its colormap must have the same visual. The colormap's visual is specified as an argument to the **createcolormap** operator.

8.10. colormapentrytype

A *colormapentry* is usually a single entry in a colormap; however, it may also be specified as a group of several entries (or *slots*). In such cases, a bitmask can be used to manipulate the indices of the entries and thereby derive the required color. Colormapentry objects can be accessed as dictionaries.

A colormapentry is created with the **createcolorsegment** operator. The colors of a colormapentry are accessed with **putcolor** and **getcolor**.

A **colormapentrytype** dictionary contains the following keys:

Colormap
Mask
Slot

The value of each key is described below.

Colormap object (*read-only*)
The colormap to which the entry belongs.

Mask int (*read-only*)
A mask of bits that can be used on a multiple entry to manipulate its indices. If the entry is not a multiple entry, the value of **Mask** is 0.

Slot int (*read-only*)
An integer that is the index position of the slot in the entry. If the entry has only one slot, the value of **Slot** is 0.

8.11. cursortype

Cursor objects are composed of a *cursor image* and a *mask image*. These two images are superimposed to create the complete cursor.

Mask and cursor images each have three attributes: a font, a character within the font, and a color. The cursor image and mask image are superimposed by aligning the origins of their respective characters. This point is also the cursor *hot spot* (the pixel coordinate to which the cursor points).

You can think of the mask image as the background and the cursor image as the foreground. The mask image defines the shape and color of the background on which the cursor image is painted. The mask image is like a stencil that the cursor image is passed through; any parts of the cursor image that fall outside of the mask will not be painted. The portion of the complete cursor painted by the cursor image appears in the cursor image color. The remainder of the complete cursor appears in the mask image color. The complete cursor has a halo effect if a cursor image is superimposed on a larger mask image.

Each canvas in the hierarchy has an associated cursor object specified by its **Cursor** key; the canvas' cursor is displayed when the mouse pointer is over the canvas. When a canvas is created with the **newcanvas** operator, the new canvas inherits the cursor of its parent.

Cursors are created with the **newcursor** operator. A cursor's characters and fonts are determined by the arguments specified to the **newcursor** operator. Cursors can be accessed as dictionaries; a cursor's colors are set with two of the dictionary keys. Cursors are not guaranteed to be displayed with their specified colors because some display devices have color limitations. The mask and image are guaranteed to be painted in contrasting colors, however.

NeWS provides a special font, called **cursorfont**, that includes common cursor shapes and their corresponding masks.

A **cursortype** dictionary contains the following keys:

CursorChar
CursorColor
CursorFont
MaskChar
MaskColor
MaskFont

The value of each key is described below.

CursorChar int (*read-only*)
The integer that corresponds to the character used for the cursor image.

CursorColor object
The color with which the image is painted.

CursorFont object (*read-only*)
The font that is used for the cursor image.

MaskChar int (*read-only*)
The integer that corresponds to the character used for the mask image.

MaskColor object
The color with which the mask image is painted.

MaskFont object (*read-only*)
The font that is used for the mask image.

8.12. environmenttype

Environment objects represent information about the server run-time environment. These objects store information about input devices such as the mouse and keyboard. Each device has its own environment object that can be accessed as a dictionary; information is stored only in the subset of keys that pertain to that particular device. The environment dictionary keys are device dependent.

An environment dictionary is created with the **createdevice** operator.

An **environmenttype** dictionary contains the following keys:

BellDuration
BellPitch
BellPercent
KeyClickPercent
Leds
AutoRepeat
KeyRepeatTime
KeyRepeatThresh
MotionCompression
Threshold
AccelNumerator
AccelDenominator

The value of each key is described below.

BellDuration real *or* integer
The duration of the keyboard bell (in 2^{16} milliseconds).

BellPitch real *or* integer
The pitch of the keyboard bell (in Hz).

BellPercent real *or* integer
The volume of the keyboard bell (0.0=off, 1.0=loudest).

KeyClickPercent real *or* integer
The volume of the keyboard key click (0.0=off, 1.0=loudest).

Leds integer
The status of the keyboard LEDs (a bit mask that determines whether the LEDs are on or off).

AutoRepeat boolean
The status of keyboard auto-repeat (true=on, false=off).

KeyRepeatTime real *or* integer
The keyboard repeat key cycle time (in 2^{16} milliseconds). Determines the speed at which a key will repeat.

KeyRepeatThresh real *or* integer
The keyboard repeat key threshold (in 2^{16} milliseconds). Specifies the amount of time a key must be pressed before it begins to repeat.

MotionCompression boolean
The status of pointer motion compression (true=motion compression on, false=motion compression off). If true and the server falls behind in processing motion events, multiple events may be collapsed into one.

Threshold integer
The pointer acceleration threshold. Specifies how fast the pointer must be moved (the threshold number of pixels moved at once) before pointer acceleration takes place.

AccelNumerator real *or* integer
Specifies the numerator for the pointer acceleration multiplier. When acceleration takes place, the pointer speed will be multipied by **AccelNumerator/AccelDenominator**.

AccelDenominator real *or* integer
Specifies the denominator for the pointer acceleration multiplier. (See **Accel-Numerator** above.)

8.13. eventtype

Events are NeWS objects, generated by the system and by NeWS processes, that are used for handling input and interprocess communication. The system generates input events to report user actions such as mouse motion and key presses. The server receives information from the input devices, translates the information into NeWS events, and distributes the events to the processes that are interested in them. In addition to input events, the server also generates events that tell processes when a canvas is damaged, when an object becomes obsolete, and when a process dies while it is still referenced. NeWS lightweight processes can also generate events and submit them for distribution.

Event objects are created using the **createevent** operator. System-generated events are created automatically. Events can be accessed as dictionaries.

Events are described in detail in Chapter 3, *Events*. This section describes the keys in the event dictionary.

An **eventtype** dictionary contains the following keys:

Action
Canvas
ClientData
Exclusivity
Interest
IsInterest
IsPreChild
IsQueued
KeyState
Name
Priority
Process
Serial
TimeStamp
XLocation
YLocation
Coordinates

The value of each key is described below.

Action object
An arbitrary POSTSCRIPT language object that often depends on the value of the Name. For keystrokes, the value of **Action** is **/DownTransition** or **/UpTransition**; for mouse motion, **Action** is null.

Canvas null, canvas, dict, *or* array
In an interest, the **Canvas** key indicates the canvas whose interest list the interest is on (or null if the interest is on the pre-child interest list of the root canvas). The **Canvas** key of an interest may contain an array or dictionary; in this case, the interest is placed on the interest list of each specified canvas. When an event is expressed as an interest, this key becomes read-only.

In an event that is to be distributed, the **Canvas** key determines which canvas interest lists are searched for potential matches. If a single canvas is specified, the event is tested against that canvas' interests and the interests of that canvas' ancestors (according to the rules given in Chapter 3, *Events*). If null is specified, the event is tested against the interests of the canvas directly under the event's location (as determined by the canvas **Coordinates** key) and the interests of that canvas' ancestors. If an array or dictionary of canvases is specified, each canvas and its ancestors are considered in turn.

ClientData object
In either an interest or an event submitted for distribution, this field may hold additional information relating to the event. The server does not set or use the value of this key.

Exclusivity boolean
If the **Exclusivity** key of an interest is true, an event that matches this interest in distribution is not allowed to match any further interests. This key is meaningful only for interests; when an event is expressed as an interest, this key becomes read-only.

Interest event (*read-only*)
This read-only key is set in an event as it is distributed; its value is the interest that the event matched in order to be delivered to its recipient.

IsInterest boolean (*read-only*)
True if the event is currently on some interest list.

IsPreChild boolean
True if the event is on the pre-child interest list of its canvas(es). This key has no effect until the event is expressed as an interest; when the event is expressed as an interest, this key becomes read-only.

IsQueued boolean (*read-only*)
True if the event has been put in the input queue and has not yet been delivered.

KeyState array (*read-only*)
When keyboard translation is on, this array is empty. When translation is off, this array indicates all the keys that were down *at the time the event was distributed.* The array actually contains the **Name** values from events that had an **Action** of **/DownTransition** and that did not have a subsequent event with the same **Name** and an **Action** of **/UpTransition**. In generating this array, the test is executed before a down-event, and after an up-event, so a down-up pair with no intervening events will not be reflected in the **KeyState** array.

This key is meaningless in an interest.

Name object
An arbitrary POSTSCRIPT language object that usually indicates the kind of event. For example, keystrokes have numeric values associated with the **Name** key, corresponding to the ASCII characters (or the keys) that were pressed. Other events have name values associated with the **Name** key, such as **/Damaged** or **/EnterEvent**.

Priority number
Priority is meaningful only for interests. When an event is expressed as an interest, this key becomes read-only. Distributed events are matched against the interests expressed on a canvas in priority order, highest priority first. Among interests with the same priority, interests with the **Exclusivity** key set to true are considered first; among nonexclusive interests of the same priority, the most recently expressed interest is considered first. The default priority is 0; fractional and negative values are allowed. The priority rarely needs to be changed from its default value.

Process null *or* process
The **Process** key can be set prior to sending an event out for distribution. In a distributed event, the **Process** key restricts distribution of the event to the specified process. Distributed events usually have null in their **Process** fields and are matched against interests without restriction. The **Process** key in an interest is set by the **expressinterest** operator to be the process that will own the interest. When an event is expressed as an interest, this key becomes read-only.

Serial number (*read-only*)
The **Serial** key is read-only for both interests and events. An event's **Serial** key is automatically set to a numeric value when the event is taken off the global event queue (the value is set from a monotomically increasing counter to indicate the sequence in which the removal of events occurs). If the event is then successfully matched with an interest, the interest's **Serial** key is automatically set to the value that the event's key contains. NeWS allows an event to match an interest only when the interest's serial number is less than that of the event; this prevents an event passed to the **redistributeevent** operator from repeatedly matching the same interests before redistribution takes place.

TimeStamp number
This numeric value indicates the time an event occurred. A time value is simply the amount of time that has elapsed since the system started, calculated in units of 2^{16} milliseconds.

The current nominal resolution of a time value is 1 ms and the maximun interval is 71,582 minutes (49.7 days).

Events in the global event queue are distributed in **TimeStamp** order, and no event is delivered before the time in its **TimeStamp** field. Thus, a timer event is simply any event handed to **sendevent** with a **TimeStamp** value in the future. This key is ignored in interests.

XLocation number
System events are labeled with the cursor location at the time they are generated; this location is used to determine which canvas interest lists are tested against the event for potential matches. The location is available to recipients and is given with respect to the current transformation matrix. This key accesses the *x* coordinate of the event's location. This key is ignored in interests.

YLocation number
This key accesses the *y* coordinate of the event's location; see the explanation under **XLocation** above. This key is ignored in interests.

Coordinates [x-location y-location]
This key accesses the event's *x* and *y* locations as an array with two elements. The *x* and *y* coordinates are given with respect to the current transformation matrix.

8.14. fonttype

A NeWS *font* object is accessible as a dictionary. The **type** operator returns **dicttype** for a NeWS **font** object; the **truetype** operator returns **fonttype**.

A NeWS **font** dictionary includes all the standard keys for a POSTSCRIPT language font dictionary; it also contains the following NeWS-defined key:

WidthArray array (*read-only*)
An array of number pairs that specify the x and y components of the width of each character. The x component of character c is in **WidthArray**(2*c), and the y component is in **WidthArray**(2*c+1). The width components are given in units of the current coordinate system with respect to the origin of the character's coordinate system.

8.15. processtype

The NeWS server maintains a set of simultaneously executing lightweight *processes*. Each process object is an individual thread of control with its own graphics context, dictionary stack, execution stack, and operand stack. These lightweight processes all exist in the same address space; two processes can refer to the same object if they can both locate the object. Typically, each connection to the server obtains a separate thread of execution with its own context. A process can create, or *fork*, new processes to form a process group. Processes communicate with each other using NeWS events.

When NeWS first starts to run, it creates a single process that executes the NeWS startup file. At this time, code may be downloaded into the server and many more lightweight processes may start. The process that runs the startup file is the only process that is not created by some earlier process executing the **fork** operator.

When a process executes the **fork** operator, the newly created process is the *child* of the *parent* process that created it. The child proccess inherits its parent's dictionary stack, operand stack, and graphics state. The parent and child start out in the same process group. However, the **newprocessgroup** operator can be used to remove a process from its process group and put it in its own, new process group. Although a child process starts out with the same name space as its parent, each lightweight process can control the extent to which its name space is shared with other processes by pushing and popping dictionaries to and from its private stack.

A process can kill its child processes, or it can wait for them to die and obtain a return value from them. A process can pause to allow other processes to run. NeWS processes can also temporarily suspend themselves and other processes. A process can examine the state of other processes by opening the process objects that represent them as dictionaries.

A process dictionary contains two special keys in **systemdict**: **$error** and **errordict**. When accessed, they return the **$error** or **errordict** of the current process. To access the **$error** or **errordict** of a different process, use the corresponding magic fields in that process. Note that the **$error** field will always be private to an individual process, containing information about the last error that process encountered, but the **errordict** can be shared between processes since its reference is copied to a child process during a fork.

A **processtype** dictionary contains the following keys:

DictionaryStack
$error
errordict
ErrorCode
ErrorDetailLevel
Execee
ExecutionStack
Interests
OperandStack
ProcessName
State
Priority
Stdout
Stderr
SendContexts
SendStack

The value of each key is described below.

DictionaryStack array (*read-only*)
An array that contains the current dictionary stack of the process. The dictionary on the bottom of the stack (the **systemdict**) is array element 0, and the process' **userdict** is array element 1.

$error dict *or* null
A dictionary that contains information about the last error that the process encountered. The dictionary is filled by the **defaulterroraction** primitive when errors occur. This error dictionary is similar to the POSTSCRIPT language **$error** dictionary, but it has one additional key named **message**; if **ErrorDetailLevel** is greater than zero, **message** contains a string that describes the context of the error. If the **defaulterroraction** primitive has not been executed, the value of **$error** will be null.

errordict dict
The **errordict** that is used to resolve the process' errors. This **errordict** is copied to a forked process by the **fork** operator. The initial value of this field is a copy of the NeWS listener's **errordict**, which by default maps each error to the **defaulterroraction** operator.

ErrorCode name
A name that specifies the current errorcode of the process. This key's value is one of the following names:

accept
dictfull
dictstackoverflow
dictstackunderflow
execstackoverflow
interrupt
invalidaccess
invalidexit
invalidfileaccess
invalidfont
invalidrestore
ioerr
killprocess
limitcheck
nocurrentpoint
none
rangecheck
stackoverflow
stackunderflow
syntaxerror
timeout
typecheck
undefined
undefinedfilename
undefinedresult
unimplemented
unmatchedmark
unregistered
VMerror

Most of the error codes are standard POSTSCRIPT language error codes. However, the following five are NeWS-specific:

- **accept** indicates that something went wrong when the server tried to accept a connection from a client process.
- **killprocess** indicates that the process has been killed, usually by the **killprocess** operator.
- **none** indicates no error.
- **timeout** indicates that the process has exceeded its time quota without pausing. The NeWS **timeout** is different than the POSTSCRIPT language **timeout** because NeWS interprets **timeout** on a per process basis and each process can avoid **timeout** by using the **pause** operator.
- **unimplemented** indicates that the process has executed an operator that is not currently implemented.

ErrorDetailLevel integer
Controls the amount of detail that is included in the default error handler's error report. Setting **ErrorDetailLevel** to 0 (the default) gives a minimum of error reporting. Setting it to 1 records a more descriptive message in the **$error** dictionary, and setting it to 2 records the contents of the dictionary, execution, and operand stacks in the **$error** dictionary. The following line sets the error detail level to 1:

```
currentprocess /ErrorDetailLevel 1 put
```

Execee object (*read-only*)
The object currently being evaluated (i.e., the top of the process' execution stack).

ExecutionStack array (*read-only*)
The full current execution stack of the process, represented as an array that contains pairs of executable arrays and indices. The executable array at the bottom of the stack is element 0 of the array, and the first index is element 1. The indices indicate which element of the associated array is currently being executed.

Interests array (*read-only*)
An array that contains the current interest list of the process.

OperandStack array (*read-only*)
The full current operand stack of the process, represented as an array. The object on the bottom of the operand stack is element 0 of the array.

ProcessName string
This key can be used to store an identifying string that gives the process a name. It defaults to (Unnamed process). This value is not used by anything internal to the server but is useful for debugging. (See the manual page for `psps`.)

State array (*read-only*)
A name that specifies the current execution state of the process. The set of possible results is as follows:

- **breakpoint** indicates that the process is suspended, normally for debugging.
- **dead** indicates that the process is completely dead.
- **input_wait** indicates that the process is waiting on an event.
- **IO_wait** indicates that the process is waiting on input/output.
- **mon_wait** indicates that the process is waiting at a monitor.
- **proc_wait** indicates that the process is waiting for another process to exit.
- **runnable** indicates that the process is running.

- **zombie** indicates that the process has exited, but other processes still have references to it.

Priority int
The scheduler priority of the process. The server has an internal priority and will not execute any process whose priority is less than this value. NeWS lightweight processes whose **Priority** falls below this dynamically-changing priority limit are not scheduled to be run by the server — they are "frozen."

NeWS processes should have no need to change their priority. Process priority is only used by the X11/NeWS server to implement "grabs". Normal processes should have a priority of **UserPriority** (0). Processes that cannot block (such as NeWS support processes) should have a priority of **SystemPriority** (100).

Stdout file
The current standard output file of the process.

Stderr file
The current standard error file of the process.

SendContexts array *(read-only)*
An array that contains the current send stack of the process. The dictionary stack on the bottom of the send stack is element 0 of the array.

SendStack array *(read-only)*
Identical to **SendContexts** but in the reverse order, so that it matches the ordering of the other stacks in a process.

8.16. visualtype

A *visual* is an object that describes the permissible color properties for a canvas. Visuals are accessible as dictionaries. Each available visual is system-supplied and its dictionary is read-only. A canvas' visual can be passed as an argument to the **newcanvas** operator; the canvas then has the properties allowed by the specified visual. If no visual is specified, a default visual is used.

To obtain a list of available visuals, examine the **VisualList** key of the root canvas.

Each colormap is also associated with a visual that is specified when the colormap is created; a colormap's visual is stored in its read-only **Visual** key. Note that a colormap and its canvas must have the same visual.

A **visualtype** dictionary contains the following keys:

Size
Class
BitsPerPixel

The value of each key is described below.

Size number (*read-only*)
The maximum number of colormapentries for colormaps associated with this visual (see **colormapentrytype** and **colormaptype**).

Class number (*read-only*)
A number that indicates the color class of the visual. Visuals are divided into six classes, representing six different types of display hardware. The following list describes the classes and their effects on the mapping between pixel value and visible color. The first line of each description gives the number that is the value of the **Class** key, followed by a name (in parentheses) that is commonly used to describe that color class.

0 (StaticGray)
: The pixel value indexes a predefined, read-only colormap. For each colormap cell, the red, green, and blue values are the same, producing a gray image.

1 (GrayScale)
: The pixel value indexes a colormap that the client can alter, subject to the restriction that the red, green, and blue values of each cell must always be the same, producing a gray image.

2 (StaticColor)
: The pixel value indexes a predefined, read-only colormap. The red, green, and blue values for each cell are server-dependent.

3 (PseudoColor)
: The pixel value indexes a colormap that the client can alter. The red, green, and blue values of each cell can be selected arbitrarily.

4 (TrueColor)
: The pixel value is divided into sub-fields for red, green, and blue. Each sub-field separately indexes the appropriate primary of a predefined, read-only colormap. The red, green, and blue values for each cell are server-dependent and are selected to provide a nearly linear increasing ramp.

5 (DirectColor)
: The pixel value is divided into sub-fields for red, green, and blue. Each sub-field separately indexes the appropriate primary of a colormap that the client can alter.

BitsPerPixel number (*read-only*)
The number of bitplanes used by the canvas.

9

NeWS Operator Extensions

9

NeWS Operator Extensions

acceptconnection

listenfile **acceptconnection** file
Listens on *listenfile* for a request made by a client UNIX process for a connection with the X11/NeWS server. When the request is received, *file* connects the client to the server. Messages written by the client to the server appear on *file*; they are then sent to the server.

The *listenfile* is created by invoking **file** with the special file name `(%socketl`*n*`)`, where *n* is the IP port number used for listening.

See also: **getsocketpeername**

arccos

num **arccos** num
Computes the arc cosine in degrees of *num.*

arcsin

num **arcsin** num
Computes the arc sine in degrees of *num.*

assert

boolean errorname **assert** –
Generates a POSTSCRIPT error of type *errorname* if *boolean* is *false* .

awaitevent

– **awaitevent** event
Removes an event from the head of the current process' local input queue, then places the event on the process' operand stack. If the local input queue does not contain an event, **awaitevent** blocks until an event is placed on the queue: an event is placed on the queue when a distributed event successfully matches an interest expressed by the process.

See also: **blockinputqueue, createevent , expressinterest , redistributeevent , sendevent**

beep

– **beep** –
Generates an audible signal. On most server implementations, this rings the keyboard bell.

blockinputqueue

num *or* null **blockinputqueue** –
Inhibits distribution of events from the global event queue. When the operator is executed, a release time is calculated for the block; the release time is the sum of the current time and the argument to **blockinputqueue**. The argument can be *num* or *null*; *num* is a number in units of 2^{16} milliseconds and *null* represents a system-defined default timeout. When the operator is executed, no event is removed from the global event queue until one of the following has occurred:

- The amount of time specified by the release time has elapsed.
- The **unblockinputqueue** operator is executed.

When nested calls to **blockinputqueue** are made, no event is removed from the global event queue until one of the following has occurred:

- The amount of time specified by the greatest of the release times has elapsed.
- The **unblockinputqueue** operator has been executed once for each call to **blockinputqueue**.

Since an event used as the argument to **sendevent** is inserted in the global event queue, its distribution can be inhibited by **blockinputqueue**. However, an event used as the argument to **redistributeevent** is not inserted in the global event queue; thus, its distribution cannot be inhibited by **blockinputqueue**.
See also: **sendevent, unblockinputqueue**

breakpoint

– **breakpoint** –
Suspends the current process.

buildimage

width height bits/sample matrix proc **buildimage** canvas
Constructs a canvas object, using the *width*, *height*, *bits/sample*, and *proc* arguments as does the POSTSCRIPT language **image** operator. The parameters represent a sampled image that is a rectangular array of *width* by *height* sample values. Each value consists of *bits/sample* bits of data (1,2,4,8). The data is received as a sequence of characters (that is, 8-bit integers in the range 0 to 255). If *bits/sample* is less than 8, the sample bits are packed left to right within a character (from the high-order bit to the low-order bit). Each row is padded out to a character boundary.

The **buildimage** operator executes *proc* repeatedly to obtain the image data. The specified *proc* must place on the operand stack a string containing any number of additional characters of sample data.

If *proc* is null, **buildimage** constructs the canvas but does not initialize its contents. (This is the recommended way of creating canvases to hold offscreen images.)

The canvas object that **buildimage** creates is retained, has no parent, and is not mapped. The canvas object cannot be mapped: it can be rendered to the screen with the **imagecanvas** or **imagemaskcanvas** operators; it can also be written to a file with the **writecanvas** operator. The *matrix* argument is used to define the default coordinate system of the canvas.

See also: **imagecanvas, imagemaskcanvas , writecanvas**

canvasesunderpath

– **canvasesunderpath** array

Returns a nested array of canvases that "intersect" the current path, starting with the current canvas. A canvas "intersects" the path if the canvas itself or any of its children fall within the area described by the path. Both opaque and transparent canvases can intersect the path. An opaque canvas can also "consume" the path; that is, prevent any younger siblings that it visually obscures from themselves intersecting the path. A transparent canvas cannot consume the path.

The returned array has the following format:

[*parent* [*child* [..] *child* [..] ..]]

The array is a nested array whose first element is the parent canvas that either intersects the path or has one or more children that themselves intersect the path. The second element is an array whose elements are the children that intersect the path. If a child itself has children that intersect the path, those children appear in a subarray in the position immediately after the child itself.

Note the following examples:

- No canvas intersects the current path:

  []

- The current canvas (`A`) intersects the current path:

  ```
  [A [] ]
  ```

- The current canvas (`A`) and one of its children (`B`) intersect the current path:

  ```
  [A [B [] ] ]
  ```

- The current canvas (`A`) and two of its children (`B` and `C`) intersect the current path:

  ```
  [A [B [] C [] ] ]
  ```

- A canvas (`A`), its child (`B`), and grandchild (`C`) intersect the current path:

  ```
  [A [B [C [] ] ] ]
  ```

- A canvas (`A`), three children (`B`, `E`, and `F`), and three grandchildren (`C`, `D`, and `G`) intersect the current path:

  ```
  [A [B [C [] D [] ] E [] F [G [] ] ] ]
  ```

canvasesunderpoint

x y *or* null **canvasesunderpoint** array
Returns an array containing the canvas under the given point and all the canvas' ancestors. The ordering is from leaf to root; thus, the canvas under the point is the first canvas in the array, and the root canvas is the last canvas in the array. If *null* is specified instead of *x*, *y*, the operator returns the hierarchy of the canvas that was under the cursor position when the last event was distributed from the global input queue, provided that the event contained meaningful cursor coordinates.

NOTE *This operator does not return canvases that lie geometrically under the given point. The operator describes a canvas' ancestry, returning its parent canvas, its grandparent canvas, and so forth. This can be used to determine how default event distribution takes place from a given canvas.*

See also: **currentcursorlocation**

canvastobottom

canvas **canvastobottom** –
Moves *canvas* to the bottom of its list of siblings.

See also: **insertcanvasbelow**

canvastotop

canvas **canvastotop** –
Moves *canvas* to the top of its list of siblings.

See also: **insertcanvasabove**

clearsendcontexts

– **clearsendcontexts** –
Removes all history of currently executing send contexts from the current process. This means that the dictionary stack will not revert to its previous state when exiting from the currently executing send context(s).

This operator is useful when no return from a **send** is possible, as in a forked process.

See also: **send**

clipcanvas

– **clipcanvas** –
The **clipcanvas** operator is identical to **clip**, except that it sets a clipping path that is an attribute of the current canvas, rather than of the current graphics state. The operator imposes clipping restrictions on all painting operations aimed at the current canvas. This is typically used during damage repair to restrict update operations to the damaged region. If the current path is empty, **clipcanvas** removes the clipping restriction of the current canvas, if such a restriction exists. Note that **clipcanvas** does not intersect the current path with the existing canvas clipping region, as the **clip** operator does.

The clipping boundary set by this operator is not affected by **initgraphics**, **initclip**, **gsave**, **grestore**, or any of the other graphics state modifiers. Graphics operations are clipped to the intersection of the canvas clip, the graphics state clip, and the shape of the canvas.

The clipping path set by this operator is not the clipping path manipulated by the operations **clip, clippath, eoclip,** and **initclip**. The **initclip** operator sets its clipping path to the shape of the canvas.

See also: **damagepath, clipcanvaspath**

clipcanvaspath

– **clipcanvaspath** –

Sets the current path to be the clipping path for the current canvas as set by **clipcanvas.**

continueprocess

process **continueprocess** –

Restarts a suspended process.

See also: **suspendprocess, breakpoint**

contrastswithcurrent

color **contrastswithcurrent** boolean

Returns *true* if the *color* argument is different from the current color; otherwise, returns *false* .

This operator takes into account the characteristics of the current device. Boolean operators, such as **eq**, can be used to compare colors without accounting for the current device.

copyarea

dx dy **copyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. The non-zero winding number rule is used to define the inside and outside of the path.

NOTE *This primitive might be used to scroll a text window.*

countfileinputtoken

file **countfileinputtoken** integer

Returns the number of usertokens associated with the given file, ignoring null tokens at the end of the list. (Normally, the returned number is simply the number of user tokens that have been defined, since applications rarely define null user tokens.) The returned index can be used as the next slot into which a user token can be stored.

countinputqueue

– **countinputqueue** num

Returns the number of events currently available from the process' local input queue.

createcolormap

visual **createcolormap** cmap
Returns an empty colormap for the specified visual.

createcolorsegment

cmap color **createcolorsegment** cmapseg
cmap C P **createcolorsegment** cmapsegs
In the first syntactic form, *cmap* is a colormap and *color* is a NeWS color object. The operator returns a single colorsegment of one entry. If the specified colormap is static, the entry returned is the one that has the closest match to the specified color value. If the colormap is dynamic, a new entry is set to the specified color value, unless the colormap is full, in which case the entry returned is the one that most closely matches the specified color.

In the second syntactic form, *cmap* is a colormap; both *C* and *P* are integers. *C* represents the number of colorsegments to be returned. *P* represents the number of planes to be used in the mask of each returned colorsegment.

createdevice

string **createdevice** boolean *or* canvas *or* env
Creates and initializes a new device, such as a framebuffer, keyboard, or mouse. The *string* argument, which is system dependent, indicates the device to be initialized. For example, the strings `/dev/fb`, `/dev/keyboard`, and `/dev/mouse` might represent a framebuffer, keyboard, and mouse.

If **createdevice** fails to create the specified device, it returns *false* . If it succeeds, it returns the specified device. If a framebuffer was specified, the returned device is an object of type **canvas**. If an input device, such as a keyboard or mouse, was specified, the returned device is an object of type **environment**. The returned device is system and implementation dependent.

This operator should only be called during system initialization.

createevent

– **createevent** event
Creates an object of type **event** and initializes its fields to either null or zero.
See also: **awaitevent, redistributeevent , expressinterest , sendevent**

createmonitor

– **createmonitor** monitor
Creates a new monitor object.
See also: **monitorlocked, monitor**

createoverlay

canvas **createoverlay** overlaycanvas
Creates a new canvas that is an *overlay canvas* and is associated with the non-overlay canvas specified by the *canvas* argument.

An overlay canvas can only be created over an existing non-overlay canvas and is always transparent. However, when graphic objects are drawn on an overlay, they appear on the overlay itself, rather than on the canvas below. Overlays are intended for use in transient or animated drawing procedures, such as the creation of *rubber-band* boxes, which expand or contract according to mouse movement, such as when a user is resizing a window.

See Chapter 2, *Canvases* for further information on overlays.

currentautobind

– **currentautobind** boolean

Returns *true* or *false* , depending on whether or not autobinding is enabled for the current process.

NOTE *When the POSTSCRIPT language interpreter encounters an executable name, it searches the dictionary stack from the top to the bottom until it finds a definition for this name. This procedure allows programmers to redefine names selectively; each name can be redefined in a dictionary placed on the dictionary stack above the normal name definition.*

However, the procedure also means that execution time tends to increase in proportion to the size of the dictionary stack. To alleviate this problem, the POSTSCRIPT language provides an operator named **bind** *that circumvents the lookup process. The operator examines the contents of a specified procedure and checks each executable name that it encounters. If a name resolves to an operator object in the context of the current dictionary stack,* **bind** *modifies the procedure by replacing the encountered name with the associated operator object. This has the effect of eliminating the time required by name lookups when the procedure is executed. Note, however, that it also removes the flexibility of being able to change a procedure's behavior by redefining names prior to execution.*

When autobinding is enabled, the effect is as if the **bind** *operator were called automatically in every procedure.*

See also: **setautobind**

currentbackcolor

– **currentbackcolor** color

Returns the background color, which is the color painted by **erasepage**.

See also: **setbackcolor**

currentbackpixel

– **currentbackpixel** integer

Returns an integer that is an index into a colormap and corresponds to the current color of the background.

See also: **setbackpixel**

currentcanvas

– **currentcanvas** canvas

Returns the current value of the canvas parameter in the graphics state.

currentcolor

– **currentcolor** color
Returns the current color as set by **setcolor**, **setrgbcolor**, **sethsbcolor**, or **setpixel**.

currentcursorlocation

– **currentcursorlocation** x y
Returns the position that was occupied by the cursor when the last event was distributed from the global input queue, provided that the event contained meaningful cursor coordinates.
See also: **canvasesunderpoint**

currentfontmem

– **currentfontmem** num
Returns the size of the font memory cache in units of kilobytes: this is the amount of memory that is used to store unused fonts in the system. For an explanation of this cache and its use, see See Chapter 7, *Memory Management.*
See also: **setfontmem**

currentpacking

– **currentpacking** bool
Returns the current array-packing mode.
See also: **packedarray, setpacking**

currentpath

– **currentpath** path
Returns an object of type *path* that describes the current path.

currentpixel

– **currentpixel** integer
Returns an integer that is an index into a colormap and corresponds to the current color of the graphics context.

currentplanemask

– **currentplanemask** integer
Returns the integer currently used as the planemask. The pixel value used by the current graphics context is AND'd with the planemask during drawing operations.
See also: **setplanemask**

currentprintermatch

– **currentprintermatch** boolean
Returns the current value of the **printermatch** flag in the graphics state.
See also: **setprintermatch**

currentprocess

– **currentprocess** process
Returns an object that represents the current process.

currentrasteropcode

– **currentrasteropcode** num
Returns a number that represents the current rasterop combination function. See **setrasteropcode** for a table of the rasterop combination functions and a discussion of its use.

See also: **setrasteropcode**

currentshared

– **currentshared** boolean
Returns *true* or *false* , depending on whether the current allocation status of the current process from the shared VM pool is enabled or disabled. See **setshared** for an explanation of the shared VM pool.

See also: **setshared**

currentstate

– **currentstate** state
Returns a **graphicsstate** object that is a snapshot of the current graphics state.

See also: **setstate**

currenttime

– **currenttime** num
Returns a time value *n.nnn* (in units of 2^{16} milliseconds) that represents time elapsed since some unspecified starting time.

This operator is guaranteed only as follows: the difference of the results of two successive calls is approximately the time that has elapsed between the calls.

damagepath

– **damagepath** –
Sets the current path to be the damage path of the current canvas. The damage path will be cleared.

The *damage path* represents those parts of the canvas that have been damaged and cannot be repainted from stored bitmaps. Processes can arrange to be notified of damage by expressing interest in *damage events.* When damage occurs to a canvas, a damage event is generated by the server.

See also: **clipcanvas**

defaulterroraction

any errorname **defaulterroraction** –
Produces an **$error** dictionary for the current process as if the error specified by *errorname* had been encountered while executing the object *any*. The operator will then execute the **stop** primitive.

NOTE *These actions are similar to the actions of the default error handling procedures described in the PostScript Language Reference Manual.*

emptypath

– **emptypath** boolean

Returns *true* if the current path is empty, otherwise *false* .

encodefont

font array **encodefont** font
font name **encodefont** font

If the *array* argument is specified, this operator creates a new font that is identical to the original font specified by the *font* argument, except that the **/Encoding** array of the old font is replaced by the specified *array* argument.

If the *name* argument is specified, the font bearing that name is located in the encoding directory and is encoded.

eoclipcanvas

– **eoclipcanvas** –

This is the same as **clipcanvas**, except that it uses the even-odd rule, rather than the non-zero winding number rule.

See also: **clipcanvas**

eocopyarea

dx dy **eocopyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. The even-odd rule is used to define the inside and outside of the path.

NOTE *This primitive might be used to scroll a text window.*

See also: **copyarea**

eoreshapecanvas

canvas **eoreshapecanvas** –

The **eoreshapecanvas** operator is identical to **reshapecanvas**, except that it uses the even-odd rule to interpret the path.

See also: **reshapecanvas**

eoextenddamage

– **eoextenddamage** –

Adds the current path to the damage shape for the current canvas. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. This operator uses the even-odd rule.

eoextenddamageall

– **eoextenddamageall** –

Adds the visible parts of the current path to the damage shape for the current canvas and the damage shapes of its children. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. The **eoextenddamageall** operator uses the even-odd rule.

eowritecanvas

file *or* string **eowritecanvas** –

This operator is identical to **writecanvas**, except that **eowritecanvas** uses the even-odd rule to define the path.

See also: **writecanvas, writescreen , eowritescreen**

eowritescreen

file *or* string **eowritescreen** –

This operator is identical to **writescreen**, except that **eowritescreen** uses the even-odd rule to define the path.

See also: **writecanvas, writescreen , eowritecanvas**

expressinterest

event **expressinterest** –

event process **expressinterest** –

Expresses interest in receiving an event distributed from the global event queue. If a *process* argument is specified, interest is expressed on behalf of that process; otherwise, interest is expressed on behalf of the current process.

When passed to **expressinterest**, the *event* becomes an interest, against which each event distributed from the global event queue is compared. When a distributed event matches the interest, a copy of the distributed event is placed on the process' local input queue.

If the *event* argument is already an interest, the **expressinterest** operator takes no action when called.

See also: **awaitevent, createevent , redistributeevent , revokeinterest , sendevent**

extenddamage

– **extenddamage** –

Adds the current path to the damage shape for the current canvas. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. This operator uses the non-zero winding number rule.

extenddamageall

– **extenddamageall** –

Adds the visible parts of the current path to the damage shape for the current canvas and the damage shapes of its children. A *damage* event is distributed if damage was not present on a particular canvas. This operator uses the non-zero winding number rule.

See also: **eoextenddamageall**

file

string1 string2 **file** file

Creates a *file* object for the file identified by *string1*, accessing it as specified by *string2*. This operator is the same as the standard POSTSCRIPT language version, except that a specific search procedure is used to locate existing files. The **file** operator first tries to open *string1* in the current directory (*./string1*). If that fails, it tries to locate and open *string1* in the home directory (*~/string1*). If that fails, it tries to open `$OPENWINHOME/etc/`*string1*.

The **file** operator can be used to create files for connections between client processes and the NeWS server; these files are socket connections and are given special filenames. Files that listen for a connection from some other process have the special filename `(%socketl`*n*`)`, where *n* is the port number that is used for listening. Files that establish a connection between two processes have either the filename `(%socketc`*n*`)` or the filename `(%socketc`*n.h*`)`, where *n* is the port number and *h* is the hostname. The connection file looks for a listener file on port *n* and host *h* (the default host is the local host); if it finds the specified listener file, it establishes the connection.

The **file** operator can also be used to run programs and connect their input or output to a *file* object. Opening the special filename `(%pipe`*command*`)` executes the UNIX command by passing it to the shell. Either the standard input or output is connected to the returned *file*, depending on whether *string2* is `(w)` or `(r)`.

findfilefont

string **findfilefont** font
Reads the font family file named by *string* and returns a newly-created font object that refers to it. The font is entered into the **FontDirectory** under the font name in the family file.

NOTE *This operator allows a bitmap font to be loaded after start-up has already occurred.*

fontascent

font **fontascent** number
Returns the specified *font*'s ascent, which is the logical distance that a character in the font extends above the baseline. Specific characters may extend beyond this distance. The measurement is given in units of the current coordinate system.

fontdescent

font **fontdescent** number
Returns *font*'s descent (as a positive number), which is the logical distance that a character in the font extends below the baseline. Specific characters may extend beyond this distance. The measurement is given in units of the current coordinate system.

fontheight

font **fontheight** number
Returns *font*'s height, which is the sum of **fontascent** and **fontdescent**.

fork

proc **fork** process
Creates a new process that executes *proc* in an environment that is a copy of the original process's environment. When *proc* exits, the process terminates. *process* is a handle by which the newly created process can be manipulated.
See also: **killprocess, killprocessgroup , waitprocess**

getcanvaslocation

canvas **getcanvaslocation** x y
Returns the location of *canvas*, relative to the current canvas. The *x,y* pair is the offset from the origin of the current coordinate system to the origin of *canvas*' default coordinate system.

getcanvasshape

– **getcanvasshape** path
Returns a path object that describes the shape of the current canvas.
See also: **movecanvas**

getcard32

string index **getcard32** integer
Returns an integer that contains the 32 bits in *string*, starting at the 32-bit word offset *index*. Note that this operator has architecture dependencies.
See also: **putcard32**

getcolor

cmapseg integer **getcolor** color
Returns the color contained in a slot of a colormapsegment. The *cmapseg* argument specifies the colormapsegment. The *integer* argument specifies the slot.
See also: **putcolor**

getenv

string1 **getenv** string2
Returns the value of the server environment variable *string1*. The value is returned as it exists in the environment of the server process; the value may be modified by **putenv** operations. The **getenv** operator fails with an `undefined` error if *string1* is not present in the environment. The **stopped** operator can be used to recover from the error.

Example

```
{ (ENV) getenv } stopped { pop (default_env_string) } if
```

See also: **putenv**

geteventlogger

– **geteventlogger** process *or* null
Returns the process that is the current event logger, or null if no such process exists.
See also: **seteventlogger**

getfileinputtoken

integer **getfileinputtoken** any
integer file **getfileinputtoken** any
Returns the object associated with the *integer* in *file*'s token list. If no *file* is specified, **currentfile** is used.

getkeyboardtranslation

– **getkeyboardtranslation** bool
Returns *true* if the kernel is interpreting the keyboard, *false* if the task is being performed by POSTSCRIPT language code.
See also: **keyboardtype, setkeyboardtranslation**

getprocesses

– **getprocesses** array
Returns an array of process groups and zombie processes. Each process group is an array of the currently active processes in the process group. Each zombie process is returned as an array containing only the zombie process, since zombie processes are not associated with any process group.

getprocessgroup

process *or* null **getprocessgroup** array
Returns the array of all processes in the process group of either the specified *process* or the current process (if *null* is specified). If *process* is a zombie process, it is the only process in the array, since zombie processes are not associated with any process group.

getsocketlocaladdress

file **getsocketlocaladdress** string
Returns a string that describes the local address of the *file* argument; this argument must be a socket file; normally, it should be a socket that is being listened to.

This operator is generally used by the server to generate a name that can be passed to client programs, telling them how to contact the server. The format of the returned string is unspecified.

getsocketpeername

file **getsocketpeername** string
Returns the name of the host to which *file* is connected. The *file* argument must be an IPC connection to another process. Such files are created with either **acceptconnection** or `(%socket) file`. This operator is normally used with **currentfile** to determine the location from which a client program is contacting the server.
See also: **acceptconnection**

harden

any **harden** any

Takes a single argument and returns it unchanged, except that if the argument is a soft reference to an object, a hard reference to the same object is returned.

See also: **soften**

hsbcolor

h s b **hsbcolor** color

Takes three numbers between 0 and 1, representing the hue, saturation, and brightness components of a color. The operator returns a *color* object that represents that color.

See also: **rgbcolor**

imagecanvas

canvas **imagecanvas** –

Renders a *canvas* onto the current canvas. This operator is similar to the **image** operator, except that the rendered image comes from a canvas, rather than from a POSTSCRIPT language procedure. When *canvas* is rendered, the unit square is transformed to the same orientation and scale as the unit square in the current transformation matrix.

The current transformation matrix can be modified (using **translate**, **scale**, or **rotate**) in order to render *canvas* to a particular area within the current canvas.

This operator maps color images onto black and white screens by dithering.

The **imagecanvas** primitive cannot be used to render a canvas into an overlay.

See also: **buildimage**, **imagemaskcanvas** , **readcanvas**

imagemaskcanvas

boolean canvas **imagemaskcanvas** –

Renders a *canvas* onto the current canvas. This operator is identical to the **imagemask** operator, except that the image comes from a canvas instead of a POSTSCRIPT language procedure. The *boolean* argument determines whether the polarity of the mask canvas is inverted.

When *canvas* is rendered, the unit square is transformed to the same orientation and scale as the unit square in the current transformation matrix.

The current transformation matrix can be modified (using **translate**, **scale**, or **rotate**) in order to render *canvas* to a particular area within the current canvas.

See also: **buildimage**, **imagecanvas** , **readcanvas**

insertcanvasabove

canvas x y **insertcanvasabove** –

Inserts the current canvas above *canvas*. The current canvas must be a sibling of *canvas*.

See also: **canvastotop**, **movecanvas**

insertcanvasbelow

canvas x y **insertcanvasbelow** –
Inserts the current canvas below *canvas*. The current canvas must be a sibling of *canvas*.
See also: **canvastobottom**

keyboardtype

– **keyboardtype** num
Returns a small integer that indicates the kind of keyboard that is attached to the server. The returned *number* is actually the return from the **KIOCTYPE** ioctl, documented under `kb`(4S).
See also: **getkeyboardtranslation, setkeyboardtranslation**

killprocess

process **killprocess** –
Kills *process*.

killprocessgroup

process **killprocessgroup** –
Kills *process* and all other processes in the same process group.
See also: **newprocessgroup**

lasteventkeystate

– **lasteventkeystate** array
Returns the **KeyState** key value of the last event delivered by the event distribution mechanism.

lasteventtime

– **lasteventtime** num
Returns the **TimeStamp** key value of the last event delivered by the event distribution mechanism.

lasteventx

– **lasteventx** num
Returns the *x* coordinate of the last event delivered by the event distribution mechanism.

lasteventy

– **lasteventy** num
Returns the *y* coordinate of the last event delivered by the event distribution mechanism.

localhostname

– **localhostname** string
Returns the network hostname of the host on which the server is running.

localhostnamearray

– **localhostnamearray** array

Returns an array whose first element is the primary hostname of the host on which the server is running, and whose remaining elements (if any exist) are aliases. The value returned by the **localhostname** operator is identical to the first element in the array returned by **localhostnamearray**.

max

a b **max** c

Compares *a* and *b* and leaves the greater of the two on the stack. Works on any data type for which **gt** is defined.

min

a b **min** c

Compares *a* and *b* and leaves the smaller of the two on the stack. Works on any data type for which **gt** is defined.

monitor

monitor procedure **monitor** –

Executes *procedure* with *monitor* locked (entered). At any given time, only one process may have a monitor locked. If a process attempts to lock a locked monitor, the process blocks until the monitor is unlocked. If an error occurs during the execution of *procedure*, and the execution stack is unwound beyond the *monitor*, the *monitor* object becomes unlocked.

See also: **createmonitor, monitorlocked**

monitorlocked

monitor **monitorlocked** boolean

Returns *true* if the *monitor* is currently locked; *false* otherwise.

See also: **createmonitor, monitor**

movecanvas

x y **movecanvas** –

x y canvas **movecanvas** –

If no *canvas* argument is specified, **movecanvas** moves the current canvas to *x,y*, relative to its parent. In this case, *x,y* is an offset from the origin of the parent canvas' default coordinate system to the origin of the current canvas' default coordinate system, measured in units of the current coordinate system.

If a *canvas* argument is specified, **movecanvas** moves *canvas* to *x,y* in the current coordinate system. In this case, *x,y* is an offset from the origin of the current coordinate system to the origin of the repositioned *canvas*' default coordinate system.

See also: **getcanvaslocation**

newcanvas

pcanvas **newcanvas** ncanvas
pcanvas visual cmap **newcanvas** ncanvas

If the *pcanvas* argument alone is specified, the operator creates a new empty canvas, *ncanvas*, whose parent is *pcanvas*. The canvas' coordinate system and shape are undefined until set with **reshapecanvas**.

The *visual* and *colormap* arguments can be used to specify a visual and a colormap for the new canvas. If these arguments are specified, the **Visual** attribute of the specified colormap must match the specified visual. If the arguments are not specified, the canvas' visual and colormap are inherited from its parent.

The new canvas defaults to being opaque if its parent is the framebuffer; transparent otherwise. The canvas defaults to being retained if it is opaque and the number of bits per pixel of the framebuffer is less than the retain threshold.

See also: **reshapecanvas**

newcursor

cursorchar maskchar font **newcursor** cursor
cursorchar maskchar cursorfont maskfont **newcursor** cursor

Creates an object of type **cursor**. Two syntactic forms can be used. With the first, a cursor is constructed using the cursor character *cursorchar* and the mask character *maskchar*; both are selected from *font*. With the second, a cursor is constructed using *cursorchar* from the font *cursorfont* and *maskchar* from the font *maskfont*. In both cases, the new cursor is initialized with a **CursorColor** value of black and a **MaskColor** value of white.

newprocessgroup

– **newprocessgroup** –

Creates a new process group, with the current process as its only member. When a process forks, the child will be in the same process group as its parent.

objectdump

file **objectdump** –

Writes to the specified file a formatted summary of the number of objects that the server has created. Nothing is returned. Note that the specified *file* must be open for writing; otherwise, an `invalidaccess` error is signaled.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

packedarray

objects n **packedarray** packedarray

Creates a packed array object of length *n*. The array contains the specified *objects* as its elements. The operator first removes the non-negative integer *n* from the operand stack. It then removes *n* objects from the operand stack, creates a packed array containing those objects, and puts the resulting packed array object on the operand stack. The resulting object is of type **packedarraytype**, has a literal attribute, and has read-only access. In all other respects, its behavior is identical to that of an ordinary array object.

See also: **currentpacking, setpacking** pathforallvec

array **pathforallvec** –
The single argument to **pathforallvec** is an array of procedures. The **path-forallvec** operator then enumerates the current path in order, executing one of the procedures in the array for each of the elements in the path. The type of the path element determines which array element will be executed. **moveto, lineto, cur-veto,** and **closepath,** are array elements 0, 1, 2, and 3, respectively. If the array is too short, **pathforallvec** tries to reduce elements of one type to another. The fifth element is used to handle conic control points. The standard POSTSCRIPT language operator **pathforall** is exactly equivalent to '4 array astore path-forallvec.'

The **pathforallvec** operator should not normally be used: the **pathforall** operator should be used instead.

pause

– **pause** –
Suspends the current process until all other eligible processes have had a chance to execute.

pointinpath

x y **pointinpath** boolean
Returns *true* if the point *x,y* is inside the current path.

postcrossings

outcanvas incanvas outname inname detailpointer? **postcrossings** –
This operator generates "crossing events", which notify the system of the movement from one canvas to another of a "state"; for example the state can be the canvas under the pointer or the focus. Examples of crossing events are **Enter** events, **Exit** events, and "focus notification" events.

The *outcanvas* argument is the canvas that the state is leaving. The *incanvas* argument is the canvas that the state is entering. Both of these arguments can be specified as either the keyword **/ReDistribute** or `null`; this is useful for managing focus states and other states that need additional modes.

The *outname* argument specifies the **Name** value of the events that indicate the canvas that the state is leaving. The *inname* argument specifies the **Name** value of the events that indicate the canvas that the state is entering. If `null` is specified for either of these arguments, generation of events with that name is suppressed.

The *detailpointer?* argument is a boolean that determines whether or not to generate extra events that indicate the relation of the state holder to the canvas under the pointer. If *detailpointer?* is set to *true* , events with an **Action** value of 5 are delivered to all canvases under the pointer that are also descendants of either *outcanvas* or *incanvas*.

The crossing events are generated in the X11 style of **Enter/Leave** notification; that is, the least common ancestor of *outcanvas* and *incanvas* is determined. (In some circumstances, there is no least common ancestor: for example, when crossing between windows on different screens; however, this is acceptable.) Events with **Name** set to *outname* are sent to *outcanvas* and to each of its ancestors up to, but excluding, the least common ancestor; these events are sent in leaf-to-root order. Then, events with **Name** set to *inname* are sent to *incanvas* and to each of its ancestors up to, but excluding, the least common ancestor, in root-to-leaf order.

The **Action** field is set to a value dependent on the canvas' position in the hierarchy with respect to *outcanvas* and *incanvas*, according to the following guidelines:

Table 9-1 *Events sent to* incanvas *and its parents*

Action	*Explanation*
0	The canvas now holds the state; the previous holder was an ancestor of this canvas.
1	The canvas is now an ancestor of the holder of the state; the previous holder was an ancestor of this canvas.
2	The canvas is now the holder of the state; the previous holder was a descendant of this canvas.
3	The canvas is now the holder of the state; the previous holder was not an ancestor or descendant of this canvas.
4	The canvas is now an ancestor of the holder of the state; the previous holder was not an ancestor or descendant of this canvas.
5	The canvas directly or indirectly contains the pointer, and is now a descendant of the holder of the state. The previous holder was not this canvas or an ancestor or descendant of it.
6	The holder of the state is now **ReDistribute**.
7	The holder of the state is now **None**.

Table 9-2 *Events sent to* outcanvas *and its parents*

Action	*Explanation*
0	The canvas used to be the holder of the state; the new holder is an ancestor of this canvas.
1	The holder of the state used to be a descendant of this canvas; the new holder is an ancestor of this canvas.
2	The canvas used to be the holder of the state; the new holder is an descendant of this canvas.
3	The canvas used to be the holder of the state; the new holder is not an ancestor or descendant of this canvas.
4	The canvas used to be an ancestor of the holder of the state; the new holder is not an ancestor or descendant of this canvas.
5	The canvas directly or indirectly contains the pointer, and used to be a descendant of the holder of the state. The new holder is not this canvas or an ancestor or descendant of it.
6	The holder of the state used to be **ReDistribute**.
7	The holder of the state used to be **None**.

This primitive is provided for the convenience of system event and state managers. An example of postcrossings usage can be found in the X11/NeWS focus manager.

putcard32

string index integer **putcard32** –

Inserts 32 bits, represented by *integer*, into the value of *string* at the 32-bit word offset specified by *index*. Note that this operator has architecture dependencies.

See also: **getcard32**

putcolor

colormapentry integer color **putcolor** –

Puts a color into a colormapentry object. The arguments to **putcolor** are a colormapentry object (which can be returned by the **createcolorsegment** operation), an integer (which is the number of the colormapentry slot into which the color is placed), and a color object. If the colormapentry object has only one slot, the value of the integer argument should be 0. Note that colormapentry must be writable to use **putcolor**.

See also: **getcolor**

putenv

string1 string2 **putenv** –

Defines the server environment variable *string1* to have the value *string2*. Environment variables inherited by the server may be modified by calls to the **putenv** operator. If the **runprogram** operator is used to create a new UNIX process, the new process inherits the server's environment variables at their current value.

See also: **getenv**

random

– **random** num

Returns a random number in the range [0,1].

readcanvas

string *or* file **readcanvas** canvas

Reads a raster file into a newly created *canvas*. The raster file can be specified either as a file or as a string that is the name of a file in the server's file name space. The created canvas is retained and opaque; it has the depth specified in the raster file, has no parent, and is not mapped. This operator sets the default coordinate system of the canvas so that the canvas' four corners correspond to the unit square.

If the specified file cannot be found, an `undefinedfilename` error is generated. If the file cannot be interpreted as a raster file, an `invalidaccess` error is generated.

Note that a canvas read into NeWS with this operator cannot be mapped to the screen; any attempt to do this results in an `invalidaccess` error. However, the canvas can be used as source for the **imagecanvas** operator.

See also: **imagecanvas, writecanvas**

recallevent

event **recallevent** –

Removes *event* from the global event queue. This primitive can be used to turn off a timer-event that has been sent but not yet delivered.

See also: **sendevent**

redistributeevent

event **redistributeevent** –

Compares against current interests an event object already returned by **awaitevent**. Comparison starts with the interest that immediately follows the successfully matched interest, which previously permitted the event object to be returned by **awaitevent**.

This operator allows an event to be matched with interests that were previously inaccessible (due, for example, to the *exclusivity* of the interest previously matched, or to the *event consumption* previously performed by some canvas).

Note that **redistributeevent** does not reinsert the event into the global event queue. No interest compared with the specified event since the last call to **sendevent** is compared with that event again.

See also: **expressinterest**

refcnt

object **refcnt** fixed fixed

Returns two numbers onto the process stack: the first is the total reference count for the specified object; the second is the soft reference count for the specified object. The counts indicate the status of the object after the operator has cleaned up the reference to the object that was given to it on the stack.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

reffinder

object **reffinder** –
object boolean **reffinder** –

Prints to standard output information on all current references to the specified object. The optional *boolean* argument can be *true* (which specifies that only information about hard references is printed) or *false* (which specifies that information about all references is printed).

If the specified object is not a counted type, a message is printed and **reffinder** returns.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

reshapecanvas

canvas **reshapecanvas** –
canvas path width **reshapecanvas** –

If a *canvas* argument alone is specified, this operator sets the shape of *canvas* to be the same as the current path, and sets *canvas*' default transformation matrix to be the same as the current transformation matrix. If *canvas* is the current canvas, an implicit **initmatrix** is performed. The entire contents of the canvas are considered to be damaged. Note that if *canvas* is the current canvas, an implicit **initclip** is performed; the **initclip** operation sets the path to the shape defined by the shape of the current canvas.

The *path* and *width* arguments can only be used if the specified *canvas* is an X canvas; if this is not so, a `typecheck` error is signaled. When the *path* and *width* arguments are specified, **reshapecanvas** can be used to give an X canvas a new border width. The *width* argument is the new border width. The *path* argument represents the drawable part of the X canvas not including the border width; it should be placed in a position inside the current path by a distance equal to *width* pixels. The following code can be used to give an X canvas a new border width.

```
canvas setcanvas
canvas bw-oldbw bw-oldbw width height rectpath
currentpath bw
newpath x y width+2*bw height+2*bw rectpath
reshapecanvas
```

where

bw	=	the new border width
oldbw	=	the old border width
width	=	the new width
height	=	the new height
x y	=	the new x and y location of the canvas in the old coordinate system

Undefined results occur if the width and paths involved are inconsistent, or do not follow the rules for X canvases.

revokeinterest

event **revokeinterest** –
event process **revokeinterest** –
Revokes an interest previously expressed either by the specified *process*, or, if no *process* argument is specified, by the current process. Following execution of this operator, no event matching *event* is distributed to the process.

Revoking interest on a non-interest has no effect.

See also: **expressinterest**

rgbcolor

r g b **rgbcolor** color
Takes three numbers between 0 and 1, respectively representing the red, green, and blue components of a color, and returns a *color* object that represents the specified color.

runprogram

string **runprogram** –
Forks a UNIX process to execute *string* as a shell command line. Standard input, standard output, and standard error are directed to `/dev/null`.

send

name object **send** –
proc object **send** –
Establishes *object*'s context by putting it and the classes in its inheritance array on the dictionary stack, executes the method, then restores the initial context. In a nested **send**, the previous send context is temporarily removed from the dictionary stack while the nested **send** executes. The *object* argument is the receiver of the message; it can be a class or an instance. In the first form, the *name* argument is the name of the method that is invoked. Any arguments required by the method must be specified; any results of the method are returned.

The second form of **send** uses a *proc* argument instead of the name of a method; *proc* is executed in the context of *object* exactly as if it had been predefined as a method and given a name that was passed to **send**.

See Chapter 4, *Classes*, for more information about classes and the **send** operator.

sendevent

event **sendevent** –
Sends an event into the event distribution mechanism. The event is positioned in the global event queue according to its **TimeStamp**. When the event at the head of the queue has a **TimeStamp** value that is less than or equal to the server's current time, the event is removed from the queue and compared with interests to find matches. Whenever a matching interest is found, the server distributes a copy of the event to the local event queue of the process with the matching interest. The process can then retrieve the event with **awaitevent**.

See Chapter 3, *Events*, for more information about event distribution.
See also: **awaitevent, createevent , recallevent , redistributeevent , expressinterest**

setautobind

boolean **setautobind** –
Enables or disables autobinding for the current process. By default, autobinding is on. (For a description of autobinding, see the entry for the **currentautobind** operator.)
See also: **currentautobind**

setbackcolor

color **setbackcolor** –
Sets the color painted by **erasepage**.
See also: **currentbackcolor**

setbackpixel

pixel **setbackpixel** –
Sets the pixel value of the background to a specified NeWS integer that is an index into a colormap. This color is used by **erasepage**.
See also: **currentbackpixel, setbackcolor**

setcanvas

canvas **setcanvas** –
Sets the current canvas to be *canvas*. Implicitly executes newpath initmatrix initclip.

setcolor

color **setcolor** –
Sets the current color to be *color*. The operation rgbcolor setcolor is identical to **setrgbcolor**; the operation hsbcolor setcolor is identical to **sethsbcolor**.

setcursorlocation

x y **setcursorlocation** –
Moves the cursor so that its hot spot is at *x*, *y* in the current canvas' coordinate system. Generates an event with **Name** set to **MouseDragged**, **Action** set to null, and the **XLocation** and **YLocation** set to the new cursor location.

seteventlogger

process **seteventlogger** –
Designates a process as an event-logger. The *process* argument must be a process that has expressed an interest (the exact nature of the interest is not significant); the process must also have entered an **awaitevent** loop. Note that the expressed interest must not match any distributed event; the interest is required to prevent **awaitevent** from returning a syntax error. The specified process becomes the *event-logger*. A copy of each event either removed from the global event queue or redistributed with **redistributeevent** will be given to this process before any other (note that the existence of the event-logger does not affect the normal running of the distribution mechanism). When the **awaitevent** loop retrieves the event copies from the event-logger's local input queue, the event-logger can proceed in whatever way is appropriate. For example, it might print certain key values in a window or to a file.

To turn off a designated event-logger, specify `null` as the argument to **seteventlogger**.

The file `eventlog.ps`, which is described in Chapter 10, *Extensibility through POSTSCRIPT Language Files*, provides a formatted display of events that can be used in the context of the **seteventlogger** operator.
See also: **geteventlogger**

setfileinputtoken

any integer **setfileinputtoken** –
any integer file **setfileinputtoken** –
Takes a specified *object* and *integer* and associates them. The object is placed in *file*'s token list at the index location specified by *integer*. If no file is specifed, **currentfile** (that is, the top file on the execution stack) is used. This operator is used to define compressed tokens for communication efficiency.

setfontmem

num **setfontmem** –
Sets the size of the font memory cache. The *num* argument specifies the size of the cache in units of kilobytes. This is the amount of memory that is used to store unused fonts in the system.
See also: **currentfontmem**

setkeyboardtranslation

boolean **setkeyboardtranslation** –
Turns the kernel translation of the keyboard on or off, according to whether the *boolean* argument is specified as *true* or *false*.
See also: **getkeyboardtranslation, keyboardtype**

setpacking

boolean **setpacking** –
Sets the array packing mode to the specified boolean value. This determines the type of executable arrays subsequently created by the POSTSCRIPT language scanner. If the specified *boolean* is *true* , packed arrays are created; if the *boolean* is *false* , ordinary arrays are created.

The array-packing mode affects the creation of procedures by the scanner when program text bracketed by { and } is encountered in the following circumstances:

- During interpretation of an executable file or string object
- During execution of the **token** operator

Note that it does not affect the creation of literal arrays by the [and] operators or by the **array** operator.

The setting continues to exist until it is overriden by a further call to **setpacking**, or undone by a call to **restore**. The packing mode is set on a per-process basis. A child process inherits the packing mode of its parent.
See also: **currentpacking, packedarray**

setpath

path **setpath** –
Sets the current path from the specified path object.

setpixel

colorobject *or* integer **setpixel** –
Sets the pixel value of the current graphics context to the specified *integer*, which is an index into a colormap.
See also: **currentpixel**

setplanemask

integer **setplanemask** –
Sets planemask to the specified integer. The pixel value used by the current graphics context is AND'd with the planemask.
See also: **currentplanemask**

setprintermatch

boolean **setprintermatch** –
Sets the current value of the **printermatch** flag in the graphics state to *boolean*. When printer matching is enabled, text output to the display is forced to be identical to text output to a printer. The metrics used by the printer are imposed on the display fonts (note that this may reduce readability). If printer matching is disabled, readability is maximized; however, the character metrics for the display do not correspond to the printer.
See also: **currentprintermatch**

setrasteropcode

num **setrasteropcode** –
Sets the current rasterop combination function, which will be used in subsequent graphics operations. The **setrasteropcode** operator accepts the following values:

Table 9-3 *Rasterop Code Values*

opcode	*function*
0	0
1	NOT (source OR destination)
2	(NOT source) AND destination
3	NOT source
4	source AND (NOT destination)
5	NOT destination
6	source XOR destination
7	NOT (source AND destination)
8	source AND destination
9	(NOT source) XOR destination
10	destination
11	(NOT source) OR destination
12	source
13	source OR (NOT destination)
14	source OR destination
15	NOT 0

The default value for the rasterop code is 12.

NOTE *The RasterOp combination function exists only to support emulation of existing window systems. It should not normally be used, since it causes problems when* *programs are used on a wide range of displays. Currently, the* **image** *primitive does not use the rasteropcode.*

See also: **currentrasteropcode**

setshared

boolean **setshared** –

Depending on the value of *boolean* (`true` or `false`), this operator either enables or disables allocations from the Shared VM pool. When a process is in shared mode, all of its allocations come from the single Shared VM pool in the server. When a process is not in shared mode, all of its allocations come from its own private VM pool. See the information on the **objectdump** operator (in Chapter 7, *Memory Management*) for an account of the types of object allocated in the server. The shared allocation status for newly forked processes is always *false* .

See also: **currentshared**

setstate

graphicsstate **setstate** –

Sets the current graphics state from *graphicsstate.*

See also: **currentstate**

shutdownserver

– **shutdownserver** –

Causes the server to exit.

soft

any **soft** boolean

Takes a single argument and returns *true* if the argument is a soft reference to an object, *false* otherwise.

soften

any **soften** any

The operator takes a single argument and returns it unchanged, except that if the argument is a reference to an object, it is returned as a soft reference. If the operator is used to soften the last existing hard reference to an object, the object becomes obsolete and an *obsolescence event* is generated by the system.

See also: **harden**

startkeyboardandmouse

– **startkeyboardandmouse** –

Initiates server processing of keyboard and mouse input. This operator is called once from the initialization file `init.ps`; it should not be called again.

suspendprocess

process **suspendprocess** –

Suspends the given process.

See also: **breakpoint, continueprocess**

tagprint

n **tagprint** –

n file **tagprint** –

Prints the integer *n* (where $-2^{15} \leq n < 2^{15}$) encoded as a tag on the specified output *file*; if no *file* is specified, prints *n* on the current output stream. Tags are used to identify packets sent from the server to client programs. See Chapter 5, *Client-Server Interface* for information on how the CPS facility uses tags.

truetype

any **truetype** name

Returns a name that identifies the true type of the object *any*. Note that this may be different from the name returned by the **type** operator: this occurs when the specified object is a *magic dictionary* that appears to be a normal dictionary or when a number with a fractional part is represented internally as a scaled integer.

typedprint

object **typedprint** –

object file **typedprint** –

Print the *object* in an encoded form on the specified output *file*; if no *file* is specified, prints *object* on the current output stream. The object can then be read by C client programs, using the CPS facility. The format in which objects are encoded is described in Chapter 5, *Client-Server Interface*.

unblockinputqueue

– **unblockinputqueue** –

Releases the input queue lock set by **blockinputqueue**. If this reduces the count of locks to 0, distribution of events from the input queue is resumed.

See also: **blockinputqueue**

undef

dictionary key **undef** –

Removes the definition of *key* from the specified *dictionary*.

vmstatus

– **vmstatus** avail used size

Returns three integers, indicating the status of memory usage. The three numbers have different meanings from the Adobe implementation. The value of *avail* is the amount of memory the server has allocated, but is not necessarily using; *used* is the amount of memory currently in use; *size* is the size of the server's heap. All sizes are in units of kilobytes.

waitprocess

process **waitprocess** value

Waits until *process* completes and returns the value that was on the top of its stack at the time of completion.

See also: **fork**

writecanvas

file *or* string **writecanvas** –

Either opens *string* as a file for writing or, if the argument is a *file*, simply writes to that file. Creates a raster file that contains an image of the region outlined by the current path in the current canvas. The **writecanvas** operator uses the non-zero winding number rule to define the path. If the current path is empty, the whole canvas is written. If the current canvas is partially obscured by one or more canvases that lie on top of it, **writecanvas** writes only the image of the current canvas.

Files written by **writecanvas** can be read by **readcanvas**; the file formats that are supported are implementation specific.

See also: **writescreen, eowritescreen , eowritecanvas**

writeobject

file object **writeobject** –

Writes the specified *object* to the specified *file*, in a readable ASCII form.

writescreen

file *or* string **writescreen** –

Either opens *string* as a file for writing or, if the argument is a *file*, simply writes to that file. Creates a raster file that contains a snapshot of the screen, clipped to the current path in the current canvas. The **writescreen** operator uses the non-zero winding number rule to define the path. If the current path is empty, the whole canvas is written. If the current canvas is partially obscured by one or more canvases that lie on top of it, **writecanvas** includes the overlapping canvases in the image.

The **writescreen** operator writes files that **readcanvas** can read; the file formats that are supported are implementation specific.

Example

This operator can be used to do a conventional screen dump, as follows:

```
framebuffer setcanvas (/tmp/snapshot) writescreen
```

See also: **writecanvas, eowritecanvas , eowritescreen**

10

Extensibility through POSTSCRIPT Language Files

10 Extensibility through POSTSCRIPT Language Files

In addition to operator and type extensions, which are part of the server itself, NeWS also supplies various POSTSCRIPT files that provide support for the NeWS programming environment; the files are loaded automatically when NeWS is initialized. You can examine these files and modify the procedures that they contain. However, if you modify them, portable NeWS programs may not run on your server.

This chapter gives an overview of the POSTSCRIPT extension files.

10.1. Initialization Files

When the NeWS server is initialized, the following extension files are automatically loaded:

`init.ps` — Initializes the frame buffer, loads most of the other initialization files described in this section, defines and starts the server, and sets various constants and system-defaults.

`redbook.ps` — Defines some POSTSCRIPT language operators that are in the *PostScript Language Reference Manual* but are not NeWS primitives.

`basics.ps` — Defines the utilities necessary to run NeWS as a filter.

`colors.ps` — Defines a set of standard colors (the same as those in the X11 `librgb` color set). Adds the dictionary **Colordict** to **systemdict**.

`cursor.ps` — Builds a dictionary useful for naming characters in **cursorfont**, which is a special font of cursors. Client-defined cursor fonts can also be built.

`statdict.ps` — Adds the **statusdict** to the **systemdict** for users needing extreme printer compatibility. The **statusdict** dictionary contains printer-specific operators such as **printername** and **setsccbatch**, as specified in Section D.6 of the *PostScript Language Reference Manual*. Many of these operators are pseudo-implemented, since they have no meaning in a window system. The file `statusdict.ps` is loaded automatically by `init.ps` at start-up.

NOTE *NeWS contains many extensions to POSTSCRIPT that do not work on printers. If you have code that you wish to send both to a NeWS server and to a printer, you should test whether the* **newcanvas** *primitive is in* **systemdict**, *since only NeWS* *servers have such an operator defined.*

`compat.ps` — Defines routines that make the server backwards-compatible with older NeWS client programs; in effect, the server is programmed to emulate previous versions of itself.

`util.ps` — Simple utilities shared by packages and NeWS applications; anything that is used by more than one package should be defined in here.

`class.ps` — Implements the NeWS class mechanism and the methods supplied by the base **Object** class.

NOTE *Some class operators are implemented in C for performance reasons; however, definitions of them in the POSTSCRIPT language are still provided.*

`rootmenu.ps` — Creates the default root menu. You can modify this in your `.user.ps` file when it is loaded.

10.2. User-Created Extension Files

This section describes files that can be created by the user; these files can contain customized NeWS initialization procedures. When NeWS is initialized, `init.ps` automatically searches for these files. The search begins in the directory from which NeWS was started; if the files are not found, the directories `~/` and `$OPENWINHOME/etc/` are searched in turn.

`.user.ps` — Contains the user's own definitions of POSTSCRIPT operators, including redefinitions of operators already in NeWS.

`.startup.ps` — Contains code fragments created by the user. If `.startup.ps` exists, its contents are executed by `init.ps` before any other package is loaded.

Other Extension Files

The following files all define extensions to NeWS; the files are loaded by individual programs rather than by `init.ps`:

`debug.ps` — Contains POSTSCRIPT procedures used for debugging.

`eventlog.ps` — Contains a small package for monitoring event distribution, described under Section 10.17, *Logging Events* below.

`journal.ps` — Contains a package for recording user actions and replaying them in *player-piano* mode, described below in Section 10.12, *Journalling Utilities*.

`repeat.ps` — Implements variable-rate repeating on keyboard keys. See Section 10.15, *Repeating Keys* below.

Extension File Contents

The following sections describe some of the most useful POSTSCRIPT procedures that are contained by the files that have been described above. Note that more exist than are documented here. All of the procedures can be customized to suit the user's individual needs.

10.3. Miscellaneous

The following operators provide miscellaneous functionality:

append

obj1 obj2 **append** obj3
Concatenates arrays, strings, and dictionaries. In the case of duplicate dictionary keys, the keys in the second dictionary overwrite those of the first.

buildsend

name *or* array object **buildsend** proc
Builds a procedure that sends a message to *object*. This procedure is self-contained and does not depend on being in a certain dictionary context. This is useful for callback procedures such as the following:

```
/myinstance /new MyClass send def
/dosomething myinstance buildsend /installcallback Manager send
```

case

value {key proc key key proc...} **case** –
Compares *value* against several keys, performing the associated procedure if a match is found. The key **/Default** matches all values.

Example

The following example uses **case** to convert a number to a string:

```
MyNumber {
   1 {(One)}
   2 {(Two)}
   3 4 5 {(Between 3 & 5)}
   /Default {(Infinity)}
} case
```

cleanoutdict

dict **cleanoutdict** –
Undefines every key in the dictionary *dict* using **undef**.

createcanvas

parentcanvas numx numy **createcanvas** canvas
Creates *canvas*, a child of *parentcanvas*, located at (0, 0) relative to its parent and possessing the given width and height.

cvad

array **cvad** dict
Considers *array* as a list of key-value pairs and fills them into a newly-created dictionary.

cvas

array **cvas** string
Converts an array of small integers into a string.

cvis

int **cvis** string
Converts a small integer into a one-character string.

dictbegin

– **dictbegin** –
Combined with **dictend**, creates a dictionary large enough for subsequent **def**s and puts it on the dictionary stack. This lets you avoid needing to guess the size of the dictionary to be created.

dictend

– **dictend** dict
Returns the dictionary created by a previous **dictbegin**; together, they "shrink-wrap" a dictionary around your **def** statements.

Example

```
/MyDict dictbegin
   /myvar 1 def
   ...
dictend def
```

dictkey

dict value **dictkey** key true *or* false
Searches *dict* for *value* and returns the corresponding key, if it is present. If *value* corresponds to several keys within *dict*, only one of the keys is returned.

fprintf

file formatstring argarray **fprintf** –
Prints to *file*.

Example

```
console (Server currenttime is:%n) [currenttime] fprintf
```

The above example prints the amount of time during which the NeWS server has been running on your console.

See also: **console**

growabledict

– **growabledict** dict
Creates a large, growable dict and leaves it on the stack.

litstring

str **litstring** str'
Replaces escapes in strings with escaped escapes.

Example

```
(blank\n) litstring
```

The above example produces the following string:

```
(\(blank\\n\))
```

modifyproc

proc {head} {tail} **modifyproc** {head proc tail}
Adds a *head* and/or a *tail* modification to a procedure, leaving on the stack an executable array that contains the modified procedure body. You can use this to override the behavior of a procedure.

Example: The following code modifies the existing version of 'myproc' by prepending the sequence '(myproc called\n) print' to the contents of the procedure each time it is invoked.

```
/myproc /myproc {(myproc called\n) print} {} modifyproc store
```

NOTE *You can use a literal name in place of any procedure you give to* **modifyproc**. *If this name is associated with a procedure in the current dictionary context, this procedure will be used in its place.*

nulloutdict

dict **nulloutdict** –
Defines every key in the dictionary *dict* to be **null**.

printf

formatstring argarray **printf** –
This is the printing form of **sprintf**. Prints on the standard output file, like **print**.
See also: **dbgprintf**

refork

processname proc **refork** –
Check to see whether a process specified by *processname* is running. If so, that process is killed with **killprocess**. Then the process (*proc*) is forked.

sendstack

– **sendstack** array

Returns the current send stack as an *array*.

sleep

interval **sleep** –

sleep sends itself an event, timestamped *interval* in the future and returns when that event is delivered.

sprintf

formatstring argarray **sprintf** string

A utility similar to the standard C `sprintf`(3S). *formatstring* is a string with '%' characters where argument substitution is to occur.

Example

```
(Here is a string:%, and an integer:%) [(Hello) 10] sprintf
```

The above example puts the following string on the stack:

```
(Here is a string:Hello, and an integer:10)
```

stringbbox

string **stringbbox** x y w h

Returns *string*'s bounding box.

See also: **fontascent, fontdescent , fontheight**

10.4. Array Operations

The following operators are provided to perform operations on arrays.

arraycontains?

array value **arraycontains?** bool

Returns the boolean true if the indicated *value* is found in the specified *array*.

arraydelete

array index **arraydelete** –

Returns a new array, deleting the value in array at position *index*. If *index* is beyond the end of the array, the last item in the newly-constructed array is deleted. Thus:

[/a /b 0 /x /y] 2 arraydelete ⇒ [/a /b /x /y]

arrayindex

array value **arrayindex** index boolean

Given an *array* and specified *value*, **arrayindex** returns the *index* of the value (if it is found) and the boolean **true**; if the *index* is not found, the operator returns no index value and the boolean **false**.

arrayinsert

array index value **arrayinsert** newarray

Creates a new array one larger than the initial array by inserting *value* at position *index*. If *index* is beyond the end of the array, *value* is appended to the end of the array. Thus:

```
[/a /b /x /y] 2 0 arrayinsert ⇒ [/a /b 0 /x /y]
```

arrayop

A B proc **arrayop** C

Performs *proc* on pairs of elements from arrays *A* and *B* in turn (for the union of the set), placing the result in array *C*.

Example

```
    [1 2 3] [4 5 6] {add} arrayop ⇒ [5 7 9]
and
    [3 4 5] [4 5 6 6] {add} arrayop ⇒ [7 9 11]
```

arrayreverse

array **arrayreverse** array_reversed

Reverses the elements of the specified *array*.

Example

```
[3 56 7 8 2 1] arrayreverse
```

The above example produces the following:

```
[1 2 8 7 56 3]
```

arrayreverseFast

array **arrayreverseFast** array_reversed

Reverses the elements of the specified *array*. It runs more quickly than **arrayreverse** but uses the operand stack; thus, it may result in stackoverflow errors.

NOTE *Stack usage is twice the array size.*

arraysequal?

array_A array_B **arraysequal?** bool

Compares the contents of the two arrays. If they are equal, it returns true, otherwise it returns false.

isarray?

any **isarray?** boolean
Returns a boolean indicating whether the object is one of the array types.

quicksort

array proc **quicksort** array
Uses the process *proc* as a rule to sort the contents of the *array*.

Example

```
[7 9 8 3 4] {gt} quicksort
```

The above example produces the following:

```
[3 4 7 8 9]
```

10.5. Conditional Operators

The following operators are provided to allow you to specify conditional operations where a value may or may not already be defined.

?get

dict key default **?get** value
If the specified *key* is found in the *dict*, its value is returned on the stack; otherwise the *default* value is returned on the stack.

?getenv

envstr defaultstr **?getenv** str
Returns the specified *envstr* as a string on the stack, if it differs from the specified *defaultstr*.

?load

key default **?load** value
Searches for the specified *key* through the dictionary stack, starting with the topmost dictionary. If the key is found, the *value* is returned on the stack; otherwise the *default* value is returned on the stack.

?put

dict key value **?put** –
Check if the *key-value* pair exists in the *dict*. If not, add the pair to the dictionary.

?undef

dict key **?undef** –
Remove the specified *key* from the dictionary, if the key is present.

10.6. Input Operators

The following operators provide functionality in the area of input and event management.

eventmgrinterest

eventname eventproc action canvas **eventmgrinterest** interest

Makes an interest that is suitable for use by **forkeventmgr** or **expressinterest**.

Example

```
/MyEventMgr [
   MenuButton {/popup MyMenu send}
   /DownTransition MyCanvas eventmgrinterest
] forkeventmgr def
```

This creates an event manager that handles popping up a menu.

forkeventmgr

interests **forkeventmgr** process

Forks a process that expresses interest in *interests*, which may be either an array whose elements are interests or a dictionary whose values are interests. Each interest must have an executable match that consumes the event returned by **awaitevent** (**eventmgrinterest** produces interests of this type)

Example

```
/MyEventMgr [
   MenuButton                      % event_name
   {/popup MyMenu send}            % event_proc
   /DownTransition                 % action
   MyCanvas                        % canvas
   eventmgrinterest                % build an interest
] forkeventmgr def
```

This invocation of **forkeventmgr** forks an event manager to watch for a /DownTransition of the MenuButton.

NOTE *The event manager uses some entries of the operand stack; do not use* **clear** *to clean up the stack in your* 'proc' *procedure.*

getanimated

x0 y0 procedure **getanimated** process

Forks a process that does animation while tracking the mouse, returning the process object *process* to the parent process. Each time the mouse moves, the process executes 'erasepage x0 y0 moveto,' pushes the current mouse coordinates *x* and *y* onto its stack, and calls *procedure*. The variables *x0*, *y0*, *x*, and *y* are available to *procedure*. After *procedure* returns, the process executes the **stroke** operator. Thus, *procedure* can use *x0*, *y0*, *x*, and *y* to build a path that is drawn each time the mouse is moved — drawing a line to the current cursor location, for example. (Note that this routine is typically useful only when the current canvas is an overlay canvas.)

The process calling *procedure* exits when the user clicks the mouse; it leaves the final mouse coordinates in an array '[x y]' on top of its stack, so that they are available to the parent process via the **waitprocess** operator. Since **erasepage** is executed each time the mouse is moved, the current canvas should be an overlay canvas when you call **getanimated**. **getanimated** is used to implement most rubber-banding operations on the screen such as in the `rubber` demo program.

See also: **createoverlay, waitprocess**

getclick

– **getclick** x0 y0

Uses **getanimated** to let the user indicate a point on the screen. **getclick** returns the location of the click on the stack.

getrect

x0 y0 **getrect** process

Uses **getanimated** to let the user "rubber-band" a rectangle with a fixed origin *x0*, *y0*. Returns a process with which you can retrieve the coordinates of the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x1 y1] (in an array) on the stack.

Example

```
100 100 getrect waitprocess
```

The above example sizes a window and then produces the following:

```
[400 432]
```

See also: **waitprocess**

getwholerect

– **getwholerect** process

Uses **getclick** and **getrect** to let the user indicate both the origin and a corner of a rectangle. Returns a process with which you can retrieve the coordinates of both the origin and the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x0 y0 x1 y1] (in an array) on the stack.

?revokeinterest

event **?revokeinterest** –

Revokes interest in an event. This operator is identical to **revokeinterest**, except that it does not generate `invalidaccesserrors` if the interest has already been revoked.

setstandardcursor

primary mask canvas **setstandardcursor** -

Sets *canvas*'s cursor to the cursor composed of the *primary* and *mask* keywords. *primary* and *mask* must be cursors in **cursorfont**, the font of standard system cursors loaded by `cursor.ps`.

Example

```
/hourg /hourg_m MyCanvas setstandardcursor
```

This sets the cursor in 'MyCanvas' to an hourglass, usually to indicate that its process will not be responding to user-input for a while.

The following table represents the cursors and their masks in **cursorfont**:

Table 10-1 *Standard NeWS Cursors*

Primary Image	*Mask Image*	*Description*	*When/Where Used*
ptr	ptr_m	arrow pointing to upper left	default
beye	beye_m	bullseye	window frame
rtarr	rtarr_m	"→" arrow	menus
xhair	xhair_m	crosshairs ("+" shape)	
xcurs	xcurs_m	"×" shape	icons
hourg	hourg_m	hourglass shape	start-up/canvas busy
nouse	nouse_m	no cursor	

See also: **setcanvascursor**

10.7. Rectangle Utilities

The following operators manage rectangular coordinates and paths; other graphics procedures are listed below, under *Graphics Utilities*.

insetrect

delta x y w h **insetrect** x' y' w' h'
Creates a new rectangle inset by *delta*.

points2rect

x y x' y' **points2rect** x y width height
Converts a rectangle specified by any two opposite corners to one specified by an origin and size.

rect

width height **rect** –
Adds a rectangle to the current path at the current pen location.

rectpath

x y width height **rectpath** –
Adds a rectangle to the current path with *x,y* as the origin.

rectsoverlap

x y w h x' y' w' h' **rectsoverlap** bool
Returns true if the two specified rectangles overlap.

rect2points

x y width height **rect2points** x y x' y'
Converts a rectangle specified by its origin and size to a pair of points that specify the origin and top right corner of the rectangle.

10.8. Class Operators

The class operators and methods are described in Chapter 4, *Classes*.

10.9. Graphics Utilities

The following operators can be used to create graphics in canvases.

colorhsb	color **colorhsb** h s b Returns the HSB values for the given color.
colorrgb	color **colorrgb** r g b Returns the RGB values for the given color.
cshow	string **cshow** – Shows *string* centered on the current location.
fillcanvas	int *or* color **fillcanvas** – Fills the entire current canvas with the gray value or color.
insetrrect	delta r x y w h **insetrrect** r' x' y' w' h' Similar to **insetrect**, but with a rounded rectangle. *See also:* **rrectpath**
ovalframe	thickness x y w h **ovalframe** – Similar to **rectframe** but with an oval.
ovalpath	x y w h **ovalpath** – Creates an oval path with the given bounding box.
polyline	array **polyline** – Draws lines using numbers from *array*. Considers *array* as an array of (*dx,dy*) pairs and then executes *dx dy* **rlineto** for each pair.
polypath	x y array **polypath** – Starts a path at (*x,y*) and then draws lines using *array* as for **polyline**. Closes the path at the end.
polyrectline	array **polyrectline** – Draws rectlinear lines using numbers from *array*. If *array* contains [*a0 a1 a2* ...], this does the equivalent of *a0* 0 **rlineto** 0 *a1* **rlineto** *a2* 0 **rlineto** and so forth.

polyrectpath

x y array **polyrectpath** –
Starts a path at (*x*,*y*) and draws rectlinear lines as for **polyrectline**. Closes the path at the end.

rectframe

thickness x y w h **rectframe** –
Creates a path composed of two rectangles, the first with origin *x*,*y* and size *w*,*h*; the second inset from this by *thickness*. Calling **eofill** fills the frame, while **stroke** creates a "wire frame" around it.

rrectframe

thickness r x y w h **rrectframe** –
Similar to **rectframe** but with a rounded rectangle.

rrectpath

r x y w h **rrectpath** –
Creates a rectangular path with rounded corners. The radius of the corner arcs is *r*, the bounding box is *x y w h*.

rshow

string **rshow** –
Shows *string* right-justified at the current location.

setshade

gray *or* color **setshade** –
Sets the current color to *gray* or to *color* value. The argument may be either a color or a shade of gray.

strokecanvas

int *or* color **strokecanvas** –
Strokes the border of the canvas with a one point edge, using the gray value or color. Currently only works for rectangular canvases.

10.10. File Access Utilities

The following operators provide file access functionality:

DefineAutoLoads

array **DefineAutoLoads** –
NeWS defines many operators that may never be used. To avoid loading the POSTSCRIPT code definition of every NeWS object at initialization, you can "lazy-define" an object to be the action of loading a file. When the object is first accessed, the file is read in; the loaded file should normally redefine the object to its original value. This form of definition is especially useful for classes, since all the methods and utility procedures that use a class can be defined in a single file, which is only read in when the class is first used. **DefineAutoLoads** takes an array of object-filename pairs.

filepathopen

filename patharray access **filepathopen** path file true *or* name false

Takes a filename, an array of path strings (such as those produced by **filepath-parse**) and the same access control string as **file**; it then tries to open *filename* in each of the paths in turn. As soon as it succeeds, it returns three values: the path that successfully located the file, a file object, and **true**. If it fails, it returns two values: *filename* and **false**.

filepathparse

pathstring **filepathparse** patharray

Takes a colon-separated set of pathnames and returns the pathnames as an array of strings.

Example

The following code parses the pathnames in the environment variable `MYPATH`; if the variable does not exist, a default set of strings is loaded.

```
/mypath (MYPATH) (.:~/bin:$OPENWINHOME/bin) ?getenv filepathparse def
```

filepathrun

filename patharray **filepathrun** path true *or* name|path false

Takes a filename and an array of path strings; the operator attempts to run the resulting file. If it cannot find the desired file in any of the given paths (using **filepathopen**), it returns *filename* and **false**. If it succeeds in finding the file, it runs the file (using **cvx exec**) in a stopped environment and reports errors. If it finds the file but cannot access or run it, **filepathrun** leaves the full path to the file and **false** on the stack. It also checks whether the file left anything on the execution stack and prints an error if this occurred. If no file was left on the stack, **filepathrun** leaves the full path to the file and **true** on the stack.

LoadFile

string **LoadFile** boolean

This is a robust, more general version of **run**. It is used to execute most NeWS startup files, returning **false** if it has problems, **true** otherwise. It searches for files in several locations: first, it prepends the user's home directory and tries to read from there; then, it passes the actual filename *string* to **file**. **file** looks first in the directory in which the X11/NeWS server was initialized, then in `$OPENWINHOME/etc`.

10.11. CID Utilities

The POSTSCRIPT files supplied by NeWS include a simple CID (Client IDentifier) synchronizer package. This generates a unique identifier used to generate a channel for client communication.

cidinterest

id **cidinterest** interest
Creates an interest appropriate for use with **forkeventmgr**. The callback procedure installed in this interest simply executes the code fragment stored in the event's **/ClientData** field.

cidinterest1only

id **cidinterest1only** interest
This is a special form of **cidinterest** that processes only one code fragment. It automatically **exits** by itself, rather than requiring the client to send the exit. For example, the *go* demo uses this operator to respond to mouse buttons which place a single stone using the above drawing fragments.

sendcidevent

id proc **sendcidevent** –
Sends a code fragment to a process created by the **cidinterest** – **forkeventmgr** usage shown above.

uniquecid

– **uniquecid** integer
Generates a unique identifier (*integer*) for use with the rest of the package.

10.12. Journalling Utilities

The following utilites allow you to control the journalling mechanism. With this mechanism, you can record and play back NeWS user input events. The file `$OPENWINHOME/demo/journaldemo` implements the following three procedures:

journalplay

– **journalplay** –
Begins replaying from the journalling file. The default filename is `/tmp/NeWS.journal`.

journalrecord

– **journalrecord** –
Starts a journalling session by opening the journalling file and logging user actions to it. The default filename is `/tmp/NeWS.journal`.

journalend

– **journalend** –
Ends a journalling session started by **journalrecord** and closes the journalling file.

Only raw mouse and keyboard events are replayed; thus, the system should be in exactly the same state at the beginning of the replay as it was at the start of the journalling session; this means that the same windows should exist in the same positions on the screen, the same user should be running the system from the same directory, and so forth. The **journalplay** operator automatically repositions the mouse to the exact position it occupied at the start of the session.

Journalling Internal Variables

The journalling utilities use the following internal variables:

- **RecordFile** — the journalling file.
- **PlayBackFile** — initially identical to **RecordFile**, this is the file from which playback takes place.
- **PlayForever** — play forever if true.
- **State** — the current state of journalling system.

These variables are explained more fully in the comments of the file `$OPENWINHOME/demo/journaldemo`. They are defined in the NeWS dictionary **journal**, created in **systemdict**.

10.13. Constants

The following constants and environment variables are provided:

console

– **console** file
Returns the file object for the system's console. Use with **fprintf** to write messages to the console.
See also: **fprintf**

framebuffer

– **framebuffer** canvas
Returns the root canvas.

minim

– **minim** real
Returns the smallest value that is representable in NeWS, which is 2^{-16}.

nulldict

– **nulldict** dict
Returns an empty, zero-length dictionary.

nullproc

– **nullproc** procedure
Returns a no-op procedure.

nullstring

– **nullstring** string
Returns an empty string.

HOME

– **HOME** string
Puts the absolute pathname to the user's home directory on the stack, or '.' if the `HOME` environment variable is not set.

OPENWINHOME

– **OPENWINHOME** string

Puts the pathname in the `OPENWINHOME` environment variable on the stack, or `/home/openwin` if this is not set. This operator is used to locate NeWS files; users should set this environment variable if they install NeWS in a non-standard location.

10.14. Key Mapping Utilities

A key may be unbound using the **unbindkey** procedure. The **bindkey** and **unbindkey** operators are described below.

bindkey

key arg **bindkey** –

Creates a new process that waits for *key* to be pressed and executes *arg* whenever that happens. If *arg* is an executable array, name, or string, it is simply handed to the PostScript interpreter. Otherwise, if it is a string, the following expression is evaluated:

{ *arg* runprogram }

Example

The following example binds the string `!make` to key F8 and assigns the NeWS-SunView selection converters to F9 and F10[2]:

```
/FunctionF8 {
    dup begin
      /Name /InsertValue  def
      /Action (!make)  def
    end
    redistributeevent
} bindkey

/FunctionF9   (sv2news_put) bindkey
/FunctionF10  (news2sv_put) bindkey
```

unbindkey

key arg **unbindkey** –

Removes the binding of the arg for the specified *key* (there is no need to call **unbindkey** before rebinding a key to a new value; the new value replaces the old in **bindkey**).

Example

The following example unbinds the key that was bound in the previous example:

```
/FunctionF9 unbindkey
```

[2] The F10 function key doesn't exist on Sun-3 keyboards.

10.15. Repeating Keys

By default, the keys of the standard typing array (which does not include the function or shift keys) repeat 20 times per second, after a .5 second threshold. The repeating keys behavior is implemented by a standalone repeat-keys package, `$OPENWINHOME/lib/NeWS/repeat.ps`, which is loaded as part of the *extended input system* started by `init.ps`. You can adjust the threshold and repeat rates according to your preference; you can do this by modifying within your `.user.ps` file the **KeyRepeatThresh** and **KeyRepeatTime** keys of your **UserProfile** dictionary. This is demonstrated by the following example:

```
UserProfile begin
  /KeyRepeatThresh              1 60 div 2 div def
  /KeyRepeatTime                1 60 div 12 div def
end
```

10.16. Standard Colors

ColorDict is a dictionary that contains named colors. It is implemented by `colors.ps`, which is loaded by `init.ps`. The color names are the same as the values in the X11R2 `librgb` library.

Example

Here are some sample color name definitions from the file:

```
/Aquamarine                 112 219 147 RGBcolor def
/MediumAquamarine           50 204 153 RGBcolor def
/Black                      0 0 0 RGBcolor def
/Blue                       0 0 255 RGBcolor def
/CadetBlue                  95 159 159 RGBcolor def
/CornflowerBlue             66 66 111 RGBcolor def
/DarkSlateBlue              107 35 142 RGBcolor def
```

RGBcolor is described as follows:

RGBcolor

red green blue **RGBcolor** color
Converts color values, specified by *red*, *green*, and *blue* values between 0 and 255, into a NeWS color object.

See also: **rgbcolor**, **setcolor**

10.17. Logging Events

The file `eventlog.ps` defines a procedure (**eventlog**) to turn logging of event distribution on and off, and a dictionary (**UnloggedEvents**), which defines those events to be excluded from the log record. "Logging" means that a copy of each event is printed as it is taken out of the event queue for distribution. This is useful for debugging the server and for clients that use events heavily. The fields of the event that are printed are **Serial**, **TimeStamp**, **Location**, **Name**, **Action**, **Canvas**, **Process**, **KeyState**, and **ClientData**.

The **Journal** application uses the event logging mechanism to allow the user to record and play back user actions. See the **journalling(1)** manual page for more information.

eventlog

bool **eventlog** –

Turns event logging on if the boolean is **true**, off if it is **false**.

The following example shows a typical log message:

```
#300 1.582 [166 161] EnterEvent 1 canvas(512x512,root,parent) null [] null
```

Log messages are sent to standard output (event logging uses the POSTSCRIPT operators **print** and ==).

UnloggedEvents

This is a dictionary of event names that are specified by the user; NeWS does not log these events. The default definition of **UnloggedEvents** is as follows:

```
/UnloggedEvents 20 dict dup begin
   /Damaged dup def
   /MouseDragged dup def
end def
```

A

NeWS Operators

A
NeWS Operators

This appendix lists all the current NeWS operators, alphabetically first, then by type.

A.1. NeWS Operators, Alphabetically

listenfile	**acceptconnection** file	listens for connection
num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
boolean errorname	**assert** –	generates an error
–	**awaitevent** event	blocks for event
–	**beep** –	generates audible signal
num	**blockinputqueue** –	blocks input events
–	**breakpoint** –	suspends current process
width height bits/sample matrix proc	**buildimage** canvas	constructs canvas object
–	**canvasesunderpath** array	returns canvases under path
x y *or* null	**canvasesunderpoint** array	returns canvases under point
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clearsendcontexts** –	removes history of send contexts
–	**clipcanvas** –	clips to canvas boundary
–	**clipcanvaspath** –	sets current path to clip
process	**continueprocess** –	restarts suspended process
color	**contrastswithcurrent** boolean	compares colors
dx dy	**copyarea** –	copies current path to *dx, dy*
file	**countfileinputtoken** integer	returns associated usertokens
–	**countinputqueue** num	returns count of input queue
visual	**createcolormap** cmap	returns colormap for visual
cmap color	**createcolorsegment** cmapseg	returns colorsegment
cmap C P	**createcolorsegment** cmapsegs	returns colorsegments
string	**createdevice** boolean, canvas *or* env	creates canvas or environment object
–	**createevent** event	creates event
–	**createmonitor** monitor	creates monitor object
canvas	**createoverlay** overlaycanvas	creates overlay canvas
–	**currentautobind** boolean	tests whether autobinding is on
–	**currentbackcolor** color	gets color painted by **erasepage**

–	**currentbackpixel** integer	returns background pixel
–	**currentcanvas** canvas	returns current canvas
–	**currentcolor** color	returns current color
–	**currentcursorlocation** x y	returns mouse coordinates
–	**currentfontmem** num	returns size of font memory cache
–	**currentpath** path	returns current path
–	**currentpixel** integer	returns index of current pixel
–	**currentplanemask** integer	returns current planemask
–	**currentprintermatch** boolean	returns **printermatch** value
–	**currentprocess** process	returns current process
–	**currentrasteropcode** num	rasterop combination function
–	**currentshared** boolean	tests whether allocation status is enabled
–	**currentstate** state	returns **graphicsstate** object
–	**currenttime** num	returns current time value
–	**damagepath** –	sets path to damage path
–	**defaulterroraction** –	produces **$error** dictionary for process
–	**emptypath** boolean	tests current path
font array	**encodefont** font	duplicates font using new encoding
font name	**encodefont** font	encodes font
–	**eoclipcanvas** –	clips to current canvas
dx dy	**eocopyarea** –	copies area to *dx, dy*
–	**eoextenddamage** –	extends damage path
–	**eoextenddamageall** –	extends damage path to all
canvas	**eoreshapecanvas** –	reshapes canvas
file *or* string	**eowritecanvas** –	writes canvas to file
file *or* string	**eowritescreen** –	writes screen to file
event	**expressinterest** –	enables reception of events
event process	**expressinterest** –	enables reception of events
–	**extenddamage** –	extends damage path
–	**extenddamgeall** –	extends damage path
string1 string2	**file** file	creates file object
string	**findfilefont** font	reads font family file, returns font
font	**fontascent** number	returns font ascent
font	**fontdescent** number	returns font descent
font	**fontheight** number	returns font height
proc	**fork** process	creates new process
canvas	**getcanvaslocation** x y	returns canvas location
–	**getcanvasshape** path	returns path object of canvas shape
string index	**getcard32** integer	returns bits from offset
cmapseg integer	**getcolor** color	returns color from colormapsegment
string1	**getenv** string2	gets value of *string1* in server
–	**geteventlogger** process	gets event logger process
integer	**getfileinputtoken** any	returns file input token
integer file	**getfileinputtoken** any	returns file input token
–	**getkeyboardtranslation** boolean	returns mode of translation
–	**getprocesses** array	returns array of process groups
process *or* null	**getprocessgroup** array	returns array of processes
file	**getsocketlocaladdress** string	returns address of file
file	**getsocketpeername** string	returns name of host connected

any	**harden** any	returns reference as hard reference
h s b	**hsbcolor** color	returns color matching *h s b*
canvas	**imagecanvas** –	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas** –	analogous to **imagemask**
canvas x y	**insertcanvasabove** –	inserts above current canvas
canvas x y	**insertcanvasbelow** –	inserts below current canvas
–	**keyboardtype** num	returns type of keyboard
process	**killprocess** –	kills process
process	**killprocessgroup** –	kills process group
–	**lasteventkeystate** array	returns **KeyState**
–	**lasteventtime** num	returns **TimeStamp**
–	**lasteventx** num	returns *x* coordinatate of event
–	**lasteventy** num	returns *y* coordinatate of event
–	**localhostname** string	returns network hostname
–	**localhostnamearray** array	returns network hostname and aliases
a b	**max** c	leaves maximum on stack
a b	**min** c	leaves minimum on stack
monitor procedure	**monitor** –	executes procedure with locked monitor
monitor	**monitorlocked** boolean	checks state of monitor
x y	**movecanvas** –	moves canvas to *x y*
x y canvas	**movecanvas** –	moves canvas to *x y*
pcanvas	**newcanvas** ncanvas	creates new canvas
pcanvas visual cmap	**newcanvas** ncanvas	creates new canvas
cursorchar maskchar font	**newcursor** cursor	creates cursor
cursorchar maskchar cursorfont maskfont	**newcursor** cursor	creates cursor
–	**newprocessgroup** –	creates new process group
file	**objectdump** –	writes summary of created objects
objects n	**packedarray** packedarray	creates packed array
array	**pathforallvec** –	analogous to **pathforall**
–	**pause** –	lets other processes run
x y	**pointinpath** boolean	tests whether point is in path
outcanvas incanvas outname inname detailpoint?	**postcrossings** –	generates events
string index integer	**putcard32** –	inserts 32 into string at offset
cmapentry int color	**putcolor** –	puts color in colormapentry
string1 string2	**putenv** –	alters value of *string1*
–	**random** num	returns random value
string	**readcanvas** canvas	reads string as canvas
event	**recallevent** –	removes event from queue
event	**redistributeevent** –	continues distribution of event
object	**refcnt** fixed fixed	returns soft and hard reference counts
object	**reffinder** –	prints references to object
canvas	**reshapecanvas** –	sets canvas to be path
canvas path width	**reshapecanvas** –	reshapes X canvas
event	**revokeinterest** –	revokes interest in event
event process	**revokeinterest** –	revokes interest in event
r g b	**rgbcolor** color	returns color object with *r g b* value
string	**runprogram** –	forks UNIX process

name object	**send** –	invokes named method in object's context
proc object	**send** –	invokes procedure in object's context
event	**sendevent** –	sends event
boolean	**setautobind** –	sets autobinding
color	**setbackcolor** –	sets **erasepage**
pixel	**setbackpixel** –	sets background pixel
canvas	**setcanvas** –	sets current canvas
color	**setcolor** –	sets current color
x y	**setcursorlocation** –	sets cursor location to *x y*
process	**seteventlogger** –	specifies process as event logger
any integer	**setfileinputtoken** –	adds object to tokenlist
any integer file	**setfileinputtoken** –	adds object to tokenlist
num	**setfontmem** –	sets size of font memory cache
boolean	**setkeyboardtranslation** –	tests whether translation is on
bool	**setpacking** –	sets packing mode
path	**setpath** –	sets path to *path*
colorobject *or* integer	**setpixel** –	sets pixel to map index
integer	**setplanemask** –	sets planemask to integer
boolean	**setprintermatch** –	sets **printermatch** flag
num	**setrasteropcode** –	sets rasterop combination function
graphicsstate	**setstate** –	sets graphics state
–	**shutdownserver** –	aborts the NeWS server
any	**soft** boolean	tests whether argument is soft reference
any	**soften** any	returns reference as soft reference
–	**startkeyboardandmouse** –	initiates server processing
process	**suspendprocess** –	suspends *process*
num	**tagprint** –	puts *num* on output stream
any	**truetype** name	identifies true type of object
object	**typedprint** –	puts *object* on output stream
–	**unblockinputqueue** –	releases input queue block
dictionary key	**undef** –	undefines *key* from *dictionary*
–	**vmstatus** avail used size	returns status of memory usage
process	**waitprocess** value	waits until completion of process
file *or* string	**writecanvas** –	writes canvas to *file*
file object	**writeobject** –	writes object to file
file *or* string	**writescreen** –	writes screen to *file*

A.2. NeWS Operators, by Functionality

The following operators are sorted according to functionality.

Canvas Operators

width height bits/sample matrix proc	**buildimage** canvas	constructs canvas object
–	**canvasesunderpath** array	returns canvases under path
x y *or* null	**canvasesunderpoint** array	returns canvases under point
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clipcanvas** –	clips to canvas boundary

–	**clipcanvaspath**	–	sets current path to clip
string	**createdevice**	boolean, canvas *or* env	creates canvas or environment object
canvas	**createoverlay**	overlaycanvas	creates overlay canvas
–	**currentcanvas**	canvas	returns current canvas
–	**eoclipcanvas**	–	clips to current canvas
canvas	**eoreshapecanvas**	–	reshapes canvas
file *or* string	**eowritecanvas**	–	writes canvas to file
file *or* string	**eowritescreen**	–	writes screen to file
canvas	**getcanvaslocation**	x y	returns canvas location
–	**getcanvasshape**	path	returns path object of canvas shape
canvas	**imagecanvas**	–	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas**	–	analogous to **imagemask**
canvas x y	**insertcanvasabove**	–	inserts above current canvas
canvas x y	**insertcanvasbelow**	–	inserts below current canvas
x y	**movecanvas**	–	moves canvas to *x y*
x y canvas	**movecanvas**	–	moves canvas to *x y*
pcanvas	**newcanvas**	ncanvas	creates new canvas
pcanvas visual cmap	**newcanvas**	ncanvas	creates new canvas
string	**readcanvas**	canvas	reads *string* as canvas
canvas	**reshapecanvas**	–	sets *canvas* to be path
canvas path width	**reshapecanvas**	–	reshapes X *canvas*
canvas	**setcanvas**	–	sets current canvas
file *or* string	**writecanvas**	–	writes canvas to *file*
file *or* string	**writescreen**	–	writes screen to *file*

Event Operators

–	**awaitevent**	event	blocks for event
num	**blockinputqueue**	–	blocks input events
–	**countinputqueue**	num	returns count of input queue
–	**createevent**	event	creates event
event	**expressinterest**	–	enables reception of events
event process	**expressinterest**	–	enables reception of events
–	**geteventlogger**	process	gets event logger process
–	**lasteventkeystate**	array	returns **KeyState**
–	**lasteventtime**	num	returns **TimeStamp**
–	**lasteventx**	num	returns *x* coordinatate of event
–	**lasteventy**	num	returns *y* coordinatate of event
outcanvas incanvas outname inname detailpoint?	**postcrossings**	–	generates events
event	**recallevent**	–	removes event from queue
event	**redistributeevent**	–	continues distribution of event
event	**revokeinterest**	–	revokes interest in event
event process	**revokeinterest**	–	revokes interest in event
event	**sendevent**	–	sends event
process	**seteventlogger**	–	specifies process as event logger
–	**unblockinputqueue**	–	releases input queue block

Mathematical Operators

num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
a b	**max** c	leaves max on stack
a b	**min** c	leaves min on stack
–	**random** num	returns random value

Process Operators

–	**breakpoint** –	suspends current process
–	**clearsendcontexts** –	removes history of send contexts
process	**continueprocess** –	restarts suspended process
–	**createmonitor** monitor	creates monitor object
–	**currentprocess** process	returns current process
–	**currentshared** boolean	tests whether allocation status is enabled
–	**defaulterroraction** –	produces **$error** dictionary for process
proc	**fork** process	creates new process
–	**getprocesses** array	returns array of process groups
process *or* null	**getprocessgroup** array	returns array of processes
process	**killprocess** –	kills *process*
process	**killprocessgroup** –	kills process group
monitor procedure	**monitor** –	executes procedure with locked monitor
monitor	**monitorlocked** boolean	checks state of monitor
–	**newprocessgroup** –	creates new process group
–	**pause** –	lets other processes run
string	**runprogram** –	forks UNIX process
process	**suspendprocess** –	suspends *process*
process	**waitprocess** value	waits until completion of process

Path Operators

dx dy	**copyarea** –	copies path to *dx, dy*
–	**currentpath** path	returns current path
–	**damagepath** –	sets path to damage path
–	**emptypath** boolean	tests current path
dx dy	**eocopyarea** –	copies area to *dx, dy*
–	**eoextenddamage** –	extends damage path
–	**extenddamage** –	extends damage path
–	**extenddamgeall** –	extends damage path
x y	**pointinpath** boolean	tests whether point is in path
path	**setpath** –	sets path to *path*

File Operators

listenfile	**acceptconnection** file	listens for connection
file	**countfileinputtoken** integer	returns associated usertokens
string1 string2	**file** file	creates file object
integer	**getfileinputtoken** any	returns file input token
integer file	**getfileinputtoken** any	returns file input token
file	**getsocketlocaladdress** string	returns address of *file*
file	**getsocketpeername** string	returns name of host connected
any integer	**setfileinputtoken** –	adds object to tokenlist
any integer file	**setfileinputtoken** –	adds object to tokenlist
num	**tagprint** –	puts *num* on output stream
object	**typedprint** –	puts *object* on output stream
file object	**writeobject** –	writes object to file

Color Operators

color	**contrastswithcurrent** boolean	compares colors
visual	**createcolormap** cmap	returns colormap for visual
cmap color	**createcolorsegment** cmapentry	returns colormapentry object
cmap color	**createcolorsegment** cmapseg	returns colorsegment
cmap C P	**createcolorsegment** cmapsegs	returns colorsegments
–	**currentbackcolor** color	gets color painted by **erasepage**
–	**currentbackpixel** integer	returns background pixel
–	**currentcolor** color	returns current color
–	**currentpixel** integer	returns index of current pixel
–	**currentplanemask** integer	returns current planemask
cmapseg integer	**getcolor** color	returns color from colormapsegment
h s b	**hsbcolor** color	returns color matching *h s b*
cmapentry int color	**putcolor** –	puts color in colormapentry
r g b	**rgbcolor** color	returns color object with *r g b* value
color	**setbackcolor** –	sets **erasepage**
pixel	**setbackpixel** –	sets background pixel
color	**setcolor** –	sets current color
colorobject *or* integer	**setpixel** –	sets pixel to map index
integer	**setplanemask** –	sets planemask to integer

Keyboard and Mouse Operators

–	**currentcursorlocation** x y	returns mouse coordinates
–	**getkeyboardtranslation** boolean	returns mode of translation
–	**getmousetranslation** boolean	tests whether events are translated
–	**keyboardtype** num	returns type of keyboard
boolean	**setkeyboardtranslation** –	tests whether translation is on
–	**startkeyboardandmouse** –	initiates server processing

Cursor Operators

–	**currentcursorlocation** x y	returns mouse coordinates
curosrchar maskchar font	**newcursor** cursor	creates cursor
cursorchar maskchar cursorfont maskfont	**newcursor** cursor	creates cursor
x y	**setcursorlocation** –	sets cursor location to *x y*

Font Operators

–	**currentfontmem** num	returns size of font memory cache
font array	**encodefont** font	duplicates font using new encoding
font name	**encodefont** font	encodes font
string	**findfilefont** font	reads font family file, returns font
font	**fontascent** number	returns font ascent
font	**fontdescent** number	returns font descent
font	**fontheight** number	returns font height
num	**setfontmem** –	sets size of font memory cache

Miscellaneous Operators

boolean errorname	**assert** –	generates an error
–	**beep** –	generates audible signal
–	**currentautobind** boolean	tests whether autobinding is on
–	**currentprintermatch** boolean	returns **printermatch** value
–	**currentrasteropcode** num	rasterop combination function
–	**currentshared** boolean	tests whether allocation status is enabled
–	**currentstate** state	returns **graphicsstate** object
–	**currenttime** num	returns current time value
string index	**getcard32** integer	returns bits from offset
string1	**getenv** string2	gets value of *string1* in server
any	**harden** any	returns reference as hard reference
–	**localhostname** string	returns network hostname
–	**localhostnamearray** array	returns network hostname and aliases
file	**objectdump** –	writes summary of created objects
objects n	**packedarray** packedarray	creates packed array
array	**pathforallvec** –	analogous to **pathforall**
string index integer	**putcard32** –	inserts bits into string at offset
string1 string2	**putenv** –	alters value of *string1*
object	**refcnt** fixed fixed	returns soft and hard reference counts
object	**reffinder** –	prints references to object
name object	**send** –	invokes named method in object's context
proc object	**send** –	invokes procedure in object's context
boolean	**setautobind** –	sets autobinding
boolean	**setpacking** –	sets packing mode
boolean	**setprintermatch** –	sets **printermatch** flag

num	**setrasteropcode**	–	sets rasterop combination function
graphicsstate	**setstate**	–	sets graphics state
–	**shutdownserver**	–	aborts the NeWS server
any	**soft**	boolean	tests whether argument is soft reference
any	**soften**	any	returns reference as soft reference
any	**truetype**	name	identifies true type of object
dictionary key	**undef**	–	undefines *key* from *dictionary*
–	**vmstatus**	avail used size	returns status of memory usage

B

The Extended Input System

B

The Extended Input System

This appendix contains information on the *Lite* user interface. This interface, previously available under NeWS 1.1, continues to be supported but will no longer be enhanced.

B.1. Building on NeWS Input Facilities

The *Extended Input System* (EIS) described in this appendix is implemented entirely in the POSTSCRIPT language on top of the basic facilities provided by the primitives in the NeWS server. It aims to support a sophisticated interface of at least the complexity of SunView or the Mac, and to provide at least one such interface as an existence proof. It also is aimed at separating independent issues in the implementation of interfaces. For example, it should be possible to provide alternatives in each of the following three categories without dependencies between categories and without requiring any change to client code:

- different input devices (1– and 3–button mice, or keyboards with different collections of function keys);
- alternative styles of input-focus, such as follow-cursor or click-to-type;
- alternative styles of selection, such as point-and-extend or wipe-through.

The EIS is sufficiently flexible that it should be possible to support a keyboard-only input system.

This chapter has several independent sections, corresponding to some of the modules of the EIS. It begins with a description of a particular user interface, implemented by the file `liteUI.ps`, which is a suggestive subset of the SunView interface. It includes a description of the requirements and facilities for a client to handle keyboard input and selections in that world.

A good deal of the processing in the EIS is carried on in a single process called "the global input handler." Some of it, however, must be done on a per-client basis; facilities are provided which are active in the client's lightweight process in the server. For example, recognizing events that indicate a change of input focus and distributing keystrokes to that focus are done in the global input handler. But recognizing user actions that indicate a selection is to be made must be done for each client, since some clients will not make selections at all, but will apply other interpretations to the same user actions.

B.2. The LiteUI Interface

The *liteUI* implementation provides distribution of keyboard input and management of selections in a style reminiscent of SunView.

Primary, Secondary, and Shelf selections are provided; Copy and Paste[3] work with all of them in the standard fashion. Selections are made when the **ViewPoint** mouse button goes down, and are always in character units. Keyboard focus may be controlled either by cursor motion into and out of windows, or by clicking a mouse button to reset the focus. In the latter mode, the **ViewPoint** (this is in **UserProfile**, and is set to **LeftMouseButton** by default) button sets both the focus and the Primary selection at the indicated position; the **ViewAdjust** (**MiddleMouseButton** by default) button restores the focus to a window, at its previous position, and without affecting the Primary selection.

There is no multi-clicking to grow a selection, and no dragging a selection with the button down. The Find and Delete functions do not yet have any clients, and so they have not been implemented. These restrictions are simply things not (yet) done in liteUI; the underlying facilities to support them are already in the EIS.

Clients of the *liteUI* interface are all lightweight processes running in the NeWS server. Such clients may have two categories of interaction with *liteUI*, getting keyboard input, and dealing with selections (for example, cutting and pasting between windows). In general, a client follows the sequence:

- In an initialization phase, the client declares its interest in various classes of activity. These classes include simple and extended keyboard input, and selection processing. In response, the EIS sets up a number of interests (some in the global input handler, some in the client's own process), and records the client in some global structures.
- The client process enters its main loop, which includes an awaitevent. Some of the events it receives will be in response to interests expressed in the initialization calls it made. These events will generally be at a high semantic level; translating mouse events into selection actions is done inside EIS. The client will typically have more work to do with these events; for example, characters may be sent across the communication channel to be processed in the client's non-POSTSCRIPT language code. Some of the processing will require calls back into EIS code; for example, a client will have to inform the system what selection it has made in response to selection events.
- Finally, when a client no longer requires various EIS facilities, it should revoke its interests, so that resources do not remain committed when it no longer needs them.

[3] These are the names of the keys on the keyboard in SunOS 4.0; however, internally the EIS refers to them as "Put" and "Get" operations.

B.3. Keyboard Input

Keyboard Input: Simple ASCII Characters

Four procedures provide access to increasingly sophisticated levels of keyboard input. The most straightforward client merely wants to get characters from the keyboard. To do this call **addkbdinterests**, passing the client canvas as an argument; then enter a loop, doing an **awaitevent** and processing the returned event.

addkbdinterests

canvas **addkbdinterests** [events]
declares the client's canvas to be a candidate for the input focus. It also creates and expresses interest in the following three kinds of events, and returns an array of the three corresponding interest-events:

ASCII Typing

The first interest has **ascii_keymap** for its **Name**, and **/DownTransition** for its **Action**. **ascii_keymap** is a dictionary provided by EIS for expressing interest in ASCII characters; it includes the translation from the user's keyboard to the ASCII character codes where that is necessary. Events which match this interest will have ASCII characters in their **Names**, and **/DownTransition** in their actions. The client can choose to see up-events too, by storing **null** into the **Action** field of this interest.

Inserting Text

The second interest has the name **/InsertValue** and a **null Action**. This will match events whose **Name** is the keyword **/InsertValue**, and whose **Action** is a string which is to be treated as though it had been typed by the user. Such events will be generated if some process is pasting selections to this window, or if function-key strings have also been requested (see below).

Input Focus

The third interest has the array [**/AcceptFocus /LoseFocus /RestoreFocus**] in its **Name.** Events matching this interest inform the client that it now has, or has lost, the input focus. These events are informational only; they do not affect the distribution of keyboard events. They are intended for clients which provide some feedback, such as a modified namestripe or a blinking caret, when they have the input focus. Clients are always free to ignore them.

Revoking Interest in Keyboard Events

A process that is about to exit, or that will continue to exist, but wants no more keyboard input, may revoke its interest in keyboard input by passing the array returned from **addkbdinterests**, along with the client canvas, to **revokekbdinterests**:

revokekbdinterests

[events] canvas **revokekbdinterests** –
Undoes all the effects of **addkbdinterests**.

Keyboard Input: Function Keys

By default, clients do not receive any events associated with function keys. A client can choose to receive function-key events, either in the form of a keyword naming the key that went down, or as a string of the form `"ESC[`*nnn*`z"` (the ASCII-standard escape sequence for such keys).

To get the function-keys identified by escape sequences, the client should pass its client canvas to **addfunctionstringsinterest**.

addfunctionstringsinterest

canvas **addfunctionstringsinterest** event
creates an interest in the function keys, expresses interest in it, and returns that event. As a result, when a function key is depressed, **awaitevent** returns an event whose **Name** is **/InsertValue**, and whose **Action** is a string holding the escape sequence defined for that key. Only function-key-down events can be received by this mechanism. **addkbdinterests** must also have been called for this procedure to have any effect.

To get the function-keys identified by name, the client should pass its client canvas to **addfunctionnamesinterest**.

addfunctionnamesinterest

canvas **addfunctionnamesinterest** event
creates an interest in the function keys, expresses interest in it, and returns that event. As a result, when a function key is pressed, **awaitevent** returns an event whose **Name** is a keyword like **/FunctionL7**.

By default, both up and down transitions on the keys are noted; the client may change this by storing **/DownTransition** (or **/UpTransition**, if that is what is desired) into the **Action** field of the returned interest. **addkbdinterests** must also have been called for this procedure to have any effect.

No special procedure is provided to revoke interests generated by either of these two procedures, since passing the interest to the **revokeinterest** primitive suffices.

Assigning Function Keys

Users may assign a procedure to be executed when a specified key goes down. See the section on **bindkey** in Chapter 10, *Extensibility through POSTSCRIPT Language Files*.

Keyboard Input: Editing and Cursor Control

If the client is passing characters through to a shell or some similar process that will do its own translations on them, it should pass them through unmodified. But if the client is dealing with text directly, it should provide the editing and caret-motion facilities defined in the user's global profile. To assist in this, the client may ask for incoming events to be checked for a match against those keyboard actions, and converted to uniform editing-events if they do. This is done by passing the client canvas to **addeditkeysinterest**.

addeditkeysinterest

canvas **addeditkeysinterest** event
creates an interest in the key combinations that are defined for global editing and caret motion, expresses interest in it, and returns that event. As a result, the client sees events with a **Name** from the set:

{Edit,Move}{Back,Fwd}{Char,Word,Field,Line,Column}

For example, here are the event names for the various **EditBack*** combinations:

/EditBackChar	Delete the character before the caret.
/EditBackWord	Delete the word before the caret.
/EditBackField	Move the caret back to the end of the preceding field if any exists, deleting its contents or selecting them in pending-delete mode.

/EditBackLine Delete from the caret back to the beginning of the current line.

/EditBackColumn Delete all characters between the caret and the nearest boundary in the line above; if the previous line ends to the left of the caret, delete back through the preceding end-of-line.

Substituting *Fwd* for *Back* indicates that the deletion or motion (see the next paragraph) extends *after* rather than before the caret. **/EditFwdLine** deletes through the next end-of-line.

Substituting *Move* for *Edit* indicates the caret is moved to the far end of the span that would be deleted by an **Edit**, but the characters are not deleted.

Again, no separate procedure is provided to revoke this interest, since the **revokeinterest** primitive does exactly what is needed.

B.4. Selections

Clients that will make selections and pass information about them to other processes declare this interest by calling **addselectioninterests**. Thereafter, EIS code will process user inputs according to the current selection policy. Occasionally, it will pass a higher-level event through to the client, when some client action is required in response. The exact interface by which a user indicates a selection is not the client's responsibility; the client must simply be prepared to handle higher-level events. Clients will also occasionally see events with a **Name** of **/Ignore**; these are events which were delivered to the client process, but handled entirely by EIS code before the event was made available to the client. The **/Ignore** event is left behind in this situation so that client code can depend on an event being on the stack when it gets control after **awaitevent** returns.

Selection Data Structures

There is no separate "selection service" in EIS; some selection processing takes place in the global input handler, and the rest in client processes. There is a global repository of data about selections, however, and there are some standard formats for the information stored in that repository and communicated between selection clients.

There are two broad styles of dealing with selections: the *communication model* and the *buffer model*. In the former, the selection holder says *"Here is my phone number, call me for answers about the selection I hold."* In the buffer model, the selection holder puts all the information about its selection into the selection-dict itself.

The selection most users are familiar with is the primary selection indicating the text they have selected in a terminal emulator window. However, there are other kinds of selection. A selection is named by its *rank*; in *liteUI*, the ranks are **/PrimarySelection**, **/SecondarySelection**, and **/ShelfSelection**.[4] For each rank, there is a dictionary containing the information known to the system about that selection. Such a dictionary will be called a *selection-dict* henceforth. It will have at least the following three keys defined:

[4] There is nothing to prevent clients from using other ranks, with names they define themselves. Strictly speaking a rank is simply a key in the **Selections** dictionary.

Table B-1 *Selection-Dict Keys*

Key	Type	Semantics
SelectionHolder	process	Which process made the selection
Canvas	The canvas	the canvas in which the selection was made.
SelectionResponder	null or process	What process will answer requests concerning this selection.

If **SelectionResponder** is defined to **null**, then the selection holder is using the buffer model, and information about the selection will be stored in other keys defined in the dictionary. setting out all available information about that selection. A few such keys have been defined because they are expected to be generally useful. These are listed in the table below. Others may be provided by clients as convenient — there is no limit on what consenting clients may say to each other kbout a selection.

Table B-2 *System-defined Selection Attributes*

Key	Type	Semantics
ContentsAscii	string	selection contents, encoded as a string
ContentsPostScript	string	selection contents, encoded as an executable POSTSCRIPT language object
SelectionObjsize	number	n >= 0; for text, 1 indicates a character
SelectionStartIndex	number	position of the first object of selection in its container
SelectionLastIndex	number	position of the last object of selection in its container

Finally, communications between clients about selections (that is, requests and their responses) are formatted as another dictionary, hereafter called a *request-dict*. When submitted by the requester, the dictionary will have a key naming each attribute for which the requestor wants a value. It may also contain commands the selection holder should execute, such as **ReplaceContents**. When received by a selection holder, a request-dict will contain the keys defined by the requester, plus the following two:

Table B-3 *Request-dict Entries*

Key	Type	Semantics
Rank	rank	the rank of selection which this request concerns
SelectionRequester	process	the process which is sending the request

The use of these various structures is detailed under the relevant event descriptions below.

Selection Procedures

This section lists the library procedures provided for clients to deal with selections.

addselectioninterests

canvas **addselectioninterests** [events]
creates and expresses interest in two classes of events, returning an array of the two interests.

The first interest matches events with names in the following list:

Table B-4 *High-Level Selection-Related Events*

/InsertValue
/SetSelectionAt
/ExtendSelectionTo
/DeSelect
/ShelveSelection
/SelectionRequest

The response required from the client to each of these events is detailed below under *Selection Events*. (Some clients may safely omit handlers for the last two; see the detailed description).

The second interest matches events which are uninteresting to the client. It arranges for EIS processing to be done by library code before the client ever sees the event.

clearselection

rank **clearselection** –
sets the indicated selection to **null**; this allows a selection holder to indicate the selection no longer exists.

selectionrequest

request-dict rank **selectionrequest** request-dict
makes a request (contained in *request-dict*) concerning the selection of the specified *rank*. The format of a *request-dict* is described above, in Table B-3, *Request-dict Entries*. The **SelectionRequester** and **Rank** entries will be filled in by **selectionrequest**.

If the **SelectionResponder** in *rank*'s selection-dict is null , then the selection holder is employing the buffer model. The **selectionrequest** procedure itself fills in the request-dict using information which the selection holder put in the selection-dict. But if the **SelectionResponder** in *rank*'s selection-dict is not null, then the selection holder is employing the communication model, and **selectionrequest** has to do a lot more work. It sends the request-dict to the **SelectionResponder** process in a **/SelectionRequest** event, and forks a process that waits for a reply. The **SelectionResponder** process is supposed to fill in the request-dict with whatever values the requester asked for, then hand back the same dictionary using **selectionresponse**; this is explained in greater detail in the description of **/SelectionRequest** below. If the **SelectionResponder** process does not respond within a certain amount of time, **selectionrequest** will return **null**.

In either case, if the indicated selection does not exist, **selectionrequest** will return **null**. Also, some keys in the request may not have an answer available. In this case they will be set to **/UnknownRequest** in the response.

selectionresponse

event **selectionresponse** –
is used by a selection holder using the request-model to return a response when it receives a selection request event. The *event* given should be the same **/SelectionRequest** event which the selection holder has just processed. (**/SelectionRequest** events are described below under *Selection Events*.) **selectionresponse** transforms *event* into a **/SelectionResponse** event and returns it to the requester.

setselection

selection-dict rank **setselection** –
is used by a process to declare itself the holder of a selection. *selection-dict* is a dictionary containing either a definition of **SelectionResponder**, or of keys which provide data about the selection itself, as described above in Table B-1, *Selection Data Structures*. *Rank* indicates which selection is being set. If another process currently holds that selection, it will be told to deselect.

getselection

rank **getselection** selection-dict
retrieves the information currently known to the system about the indicated selection. This procedure is likely to be more useful to the implementor of a package like *liteUI* than to window clients.

Selection Events

As mentioned above, clients may expect to receive six different kinds of events concerning the selection. Of these, the **/InsertValue** event has already been described under *Keyboard Input*; its usage in the selection context is exactly the same as for function strings. The remaining five events and the appropriate responses to them are presented below.

Each event is described in the following format:

EventType *short description of the event's semantics*

> **Name:**
> *keyword that identifies the event*
>
> **Action:**
> *description of the contents of the event's* **Action** *field*
>
> Response:
> *description of what the client should do when it receives such an event*

/SetSelectionAt

Informs the client the user has just made a selection in its canvas.

Name:
/SetSelectionAt

Action:

dict [	**Rank**	**/PrimarySelection | /SecondarySelection**
	X	*number*
	Y	*number*
	PendingDelete	**true | false**
	Preview	**true | false**
	Size	*number*
]		

NOTE *LiteUI provides constant values for three fields:* **PendingDelete = false, Preview = false,** *and* **Size** *= 1.*

Response:

Make a selection of the indicated **Rank** with the following parameters:

Key	*Value*
X and **Y**	indicate a position (it will be in the current canvas' coordinate system).
Size	indicates the unit to be selected; for example, in text: 0 means a null selection at the nearest character boundary, 1 corresponds to a character, and larger values indicate larger units (words, lines, etc.) whose definition is at the discretion of the client
PendingDelete	indicates whether that mode should be used (if supported by the client)
Preview	indicates whether the selection is only for feedback to the user; a selection shouldn't actually be set until a selection event is received with **Preview false**

In client POSTSCRIPT langauge code, some private processing will generally be required. For instance, the given position will have to be resolved to a character in a text window, and appropriate feedback displayed on the screen. Then the client should build a selection-dict describing the selection just made, and pass it to **setselection**, along with the rank it received in the **/SetSelectionAt** event:

```
        selection-dict rank setselection
```

'selection-dict' should contain either a non-null definition of **Selection-Responder**, or it should define keys which actually provide information about the selection (**ContentsAscii** at a minimum). In the former case, the holder is following the *communication model* of selection, and must be prepared to receive and respond to **/SelectionRequest** events as long as it holds the selection. In the latter case, the holder is following the *buffer model* of selection; requests will be answered automatically by the global input handler.

'selection-dict' will have keys added to it, so it should be created with room for at least *five* more entries beyond those defined by the client.

/ExtendSelectionTo

Informs the client the user has just adjusted the bounds of a selection in its canvas.

Name:
/ExtendSelectionTo

Action:

dict [	**Rank**	**/PrimarySelection | /SecondarySelection**
	X	*number*
	Y	*number*
	PendingDelete	**true | false**
	Preview	**true | false**
	Size	*number*

]

Response:

The dictionary in the **Action** field is the same as the **Action** of a **/SetSelectionAt** event, and the client response is very much the same. The distinction is that this event indicates a modification of an existing selection, where **/SetSelectionAt** indicates a new one.

The client should adjust the nearest end of the current selection of the indicated **Rank** to include the indicated position. If **Size** indicates growth, extend both ends as necessary to get them at a boundary of the indicated size. (For example, if **Size** has changed from 1 to 2, a text window might grow both ends of the selection to ensure that they fall at word boundaries.) Adjust the **PendingDelete** mode or ignore it as the window is editable or not.

If there was no selection of the indicated rank, pretend there was an empty one at the indicated position.

In client POSTSCRIPT language code, after doing any private processing required, processing is exactly the same as for **/SetSelectionAt**.

/DeSelect

Informs the client that it no longer holds the indicated selection.

Name:
/DeSelect

Action:
rank

Response:
Undo a selection of the given rank in this window. *Do not* call **clearselection**; the global selection information has already been updated.

/ShelveSelection

Tells the client to set the shelf selection to be the same as a selection which the client currently holds.

Name:
/ShelveSelection

Action:
rank

Response:
Buffer-model clients (those that did not define **SelectionResponder** when they set the selection) will not receive **/ShelveSelection** events; the service will copy their selection to the shelf for them. Others should set the **ShelfSelection** to be the same as the selection whose *rank* is in **Action**, using **setselection** as above.

NOTE *Be careful of the difference between the* **ShelveSelection** *and* **ShelfSelection***; the former is a selection event, and the latter is one of the selection ranks along with* **/PrimarySelection** *and* **/SecondarySelection**.

/SelectionRequest

The client is requested to provide information about a selection it holds.

Name:
/SelectionRequest

Action:
request-dict

Response:
Buffer-model clients (those that did not define **SelectionResponder** when they set the selection) will not receive **SelectionRequest** events; the service will answer the request for them.

The client should enumerate the request-dict, responding to the various requests by defining their values (as for **ContentsAscii**), or performing the requested operation (as for **/ReplaceContents**, whose value will be the replacement value). The resultant dict should be left as the **Action** of the event, which should then be passed as the argument to the procedure **selectionresponse.**

NOTE *There is no restriction on what requests may be contained in a selection request; this is left to negotiation between the requester and the selection holder. A holder may reject any request, by defining its value to be* **/UnknownRequest**.

It may be noted that there is no mechanism described here for getting a selection's contents from someplace else. In *liteUI*, user actions that precipitate such a transfer are recognized and processed in the global input handler, which then performs the selection request, and sends an **/InsertValue** event to the receiving process. The selection procedures described above provide an interface for instigating such transfers independent of user actions.

B.5. Input Focus

The input focus (where standard keyboard events are directed) is maintained by the global input-handler process, according to the current focus policy. A client becomes eligible to be the input focus by calling **addkbdinterests** (described above under *Selection Procedures*). At some later time, some user action will indicate that the client should become the focus. The client will receive an event indicating this has happened (its **Name** will be **/AcceptFocus** or **/RestoreFocus**, and its **Action** is described in the table below). Thereafter, the client will receive events whose **Names** are ASCII character codes. Loss of the keyboard focus will be indicated by the delivery of an event with **Name /LoseFocus**.

Table B-5 *Input Focus*

Name	*Action*	*Explanation*
/Restore-AcceptFocus	0	The canvas is now the focus; the previous focus was an ancestor of this canvas.
	1	The canvas is now the ancestor of the focus; the previous focus was an ancestor of this canvas.
	2	The canvas is now the focus; the previous focus was a descendant of this focus.
	3	The canvas is now the focus; the previous focus was not an ancestor or descendant of this canvas.
	4	The canvas is now an ancestor of the focus; the previous focus was not an ancestor or descendant of this canvas.
	5	The canvas directly or indirectly contains the pointer and is now a descendant of the focus. The previous canvas is not equivalent to this canvas nor is the previus canvas an ancestor or descendant of this canvas.
	6	The focus is now **PointerRoot**.
	7	The focus is now **None**.
/LoseFocus	0	The canvas used to be the focus; the new focus is an ancestor of this canvas.
	1	The canvas used to be an ancestor of the focus; the new focus is an ancestor of this canvas.
	2	The canvas used to be the focus; the new focus is a descendant of this canvas.
	3	The canvas used to be the focus; the new focus is not an ancestor or descendant of this canvas.
	4	The canvas used to be an ancestor of the focus; the new focus is not an ancestor or descendant of this canvas.

Table B-5 *Input Focus— Continued*

Name	*Action*	*Explanation*
	5	The canvas directly or indirectly contains the pointer and used to be a descendant of the focus. The new canvas is not equivalent to this canvas nor is the new canvas an ancestor or descendant of this canvas.
	6	The focus used to be **PointerRoot**.
	7	The focus used to be **None**.

This section describes a collection of routines provided to inquire about and manipulate the focus. These normally will not be called by clients of the window system; rather, they support focus-policy implementations, which then communicate with the clients.

The focus is identified in an array with two elements, a canvas and a process. The canvas will be the *canvas* argument to **addkbdinterests**. The process will be one which called **addkbdinterests**, and which should be doing an **awaitevent** for keyboard events.

setinputfocus

canvas process **setinputfocus** –
The input focus is set to be the canvas – client pair identified by the arguments to **setinputfocus**.

currentinputfocus

– **currentinputfocus** [canvas, process]
The current input focus is returned by **currentinputfocus**. If there is no current focus, **null** is returned.

hasfocus

process **hasfocus** bool
Returns **true** or **false** as the indicated process is or is not currently the input focus.

setfocusmode

keyword **setfocusmode** –
The global focus policy is reset to the policy named by the argument. Currently-supported focus policies are identified by:

/ClickFocus	As long as no function keys are down clicking the Select button will set both the focus and the primary selection in a window. Clicking Adjust will restore the focus at its last position in this window, without making any selection.
/CursorFocus	a window will receive the focus when the mouse enters its subtree, and lose it when the mouse exits. If the mouse crosses window boundaries while a function key is down, a focus change is delayed until all function keys are up, and then reflects the current situation.

/DefaultFocus events are distributed as though no EIS were in effect.

C

Omissions and Implementation Limits

C

Omissions and Implementation Limits

C.1. Operator Omissions and Limitations

The following primitives are defined in the *PostScript Language Reference Manual*. They have not been implemented in the X11/NeWS POSTSCRIPT language interpreter because they are either printer- or environment-specific.

banddevice
framedevice
renderbands
start

The following operators are implemented, but they do not do anything. If you execute them they will consume or produce the right arguments on the operand stack, but they will have no other effects. The **showpage** operator does perform an implicit **initgraphics** operation, but it otherwise has no other effect.

copypage
currentscreen
echo
setscreen
showpage

The following operators are unimplemented:

executeonly
noaccess
resetfile
reversepath

The **charpath** operator does not return the actual path outline of the string given as an argument. Instead, it returns the bounding rectangle of this path.

The forms of the **translate**, **transform**, **itransform**, and **idtransform** operators that take matrix arguments do not work. However, the two-argument forms do work. For example:

```
x y translate
```

works properly, but the following fails with a `typecheck` error:

```
x y matrix translate
```

C.2. Imaging Omissions

The only portion of the stencil/paint imaging model that is unimplemented is halftone screening. The **setscreen** and **currentscreen** operators exist, and they produce and consume appropriate objects from the operand stack, but they have no other effect. X11/NeWS uses dithering techniques instead of halftone screens.

C.3. The statusdict Dictionary

Most of the entries in the **statusdict** dictionary (described in Appendix D of the *PostScript Language Reference Manual*) are pseudo-implemented; they have reasonable values, but setting them has no effect. One exception is the job timeout. Getting and setting the job timeout will change how long a process is allowed to execute without blocking before receiving a `timeout` error.

C.4. Implementation Limits

The following table lists implementation limits of NeWS:

Table C-1 *Implementation Limits*

Quantity	*Limit*	*Explanation*
integer	32767	Integers are represented as 32 bits, 16 bits of them fraction. Integers are automatically converted to reals if they overflow.
real		Single-precision floating-point numbers are used. Reals are represented as fractional integers if they are small enough, but the type determination operators will describe them as real.
array	32767	Number of entries in an array.
dictionary	16384	Number of key/value pairs in a dictionary.
string	32767	Number of characters in a string.
name	32767	Number of characters in a name.
file		Number of open files (includes open client communication channels). The limit is `getdtablesize()`-*n*, where *n* depends on the particular server but will be about four.
userdict	250	Set by code in `init.ps`; easy to change.
operand stack	1500	Maximum size of an operand stack.
dict stack		Expanded as required.
exec stack	250	Maximum function/compound statement nesting depth.
gsave level		Expanded as required.
path		Expanded as required.
VM		The server expands to use as much VM as the underlying system permits.
interpreter level		Not applicable.
save level		Expanded as required.

C.5. Other Differences with the POSTSCRIPT Language

In addition to the omissions and differences implemented above, the POSTSCRIPT language has slightly different semantics for some standard POSTSCRIPT language operators. The NeWS versions of these operators are described in Chapter 9, *NeWS Operator Extensions* along with the wholly-new NeWS operators.

Table C-2 *NeWS Versions of Various POSTSCRIPT Language Operators*

Operator	*Note*
==	Although there is no specification for their output in the *PostScript Language Reference Manual*, you may be confused because = and == print objects out in a slightly different format than the particular implementations in the LaserWriter. = identifies dictionaries as such and prints some useful fields from the various "magic dictionary" types in NeWS. == actually prints out the first few key-value pairs in dictionaries. == uses quotation marks as follows to indicate the types of objects: '*operator-type*', '*called-operator-type*', and '*magic-variable-type!*'
bind	**bind** is implemented in NeWS, but it is useful only when autobinding is off. Autobinding is on by default. See **currentautobind** in Chapter 9, *NeWS Operator Extensions*.
file	NeWS has the additional special name '%socketl*n*' for socket-based network connections to the server. Also, the POSTSCRIPT language operators **file** and **run** (together with all the NeWS utility procedures for file access) will look for relative path names in the directory from which the server was started, *and* in `$OPENWINHOME/etc`.
vmstatus	In NeWS, **vmstatus** returns *avail*, *used*, and *size*; the *PostScript Language Reference Manual* states that it should return *level*, *used*, and *maximum*.

Index

sun microsystems

NeWS™ 2.0 Programmer's Guide

Part No: 800-2379-10
Revision A, of 25 August 1989

Systems for Open Computing™

Corporate Headquarters
Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043
415 960-1300
FAX 415 969-9131

For U.S. Sales Office locations, call:
800 821-4643
In CA: 800 821-4642

European Headquarters
Sun Microsystems Europe, Inc.
Bagshot Manor, Green Lane
Bagshot, Surrey GU19 5NL
England
0276 51440
TLX 859017

Australia: (02) 413 2666
Canada: 416 477-6745
France: (1) 40 94 80 00

Germany: (089) 95094-0
Hong Kong: 852 5-8651688
Italy: (39) 6056337
Japan: (03) 221-7021
Korea: 2-7802255
New Zealand: (04) 499 2344
Nordic Countries: +46 (0)8 7647810
PRC: 1-8315568
Singapore: 224 3388
Spain: (1) 2532003
Switzerland: (1) 8289555
The Netherlands: 033 501234

Taiwan: 2-7213257
UK: 0276 62111

Europe, Middle East, and Africa, call European Headquarters:
0276 51440

Elsewhere in the world, call Corporate Headquarters:
415 960-1300
Intercontinental Sales