NeWS™ 1.1 Manual

NeWS™ is a trademark of Sun Microsystems, Inc.
SunView™ is a trademark of Sun Microsystems, Inc.
Sun Workstation®, Sun Microsystems®, and the Sun logo
are registered trademarks of Sun Microsystems Inc.

POSTSCRIPT is a registered trademark of Adobe Systems Inc. Adobe owns copyrights related to the POSTSCRIPT language and the POSTSCRIPT interpreter. The trademark POSTSCRIPT is used herein to refer to the material supplied by Adobe or to programs written in the POSTSCRIPT language as defined by Adobe.

Macintosh is a trademark licensed to Apple Computer, Inc.

VAX is a trademark of Digital Equipment Corporation.

X Window System is a trademark of Massachusetts Institute of Technology.

UNIX is a registered trademark of AT&T Information Systems Inc.

Copyright © 1987 by Sun Microsystems, Inc.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Contents

Preface xv

Chapter 1 Introduction **3**

1.1. The Design 3

1.2. Imaging 5

1.3. Canvases 5

1.4. User Interaction — Input 6

1.5. Client Interface 6

1.6. Background and Goals 7

Device Independence 7

Portability 8

Flexibility 8

Distribution 9

Graphics 9

References 9

Chapter 2 NeWS Extension Overview **13**

2.1. The Lightweight Process Mechanism 13

2.2. Canvases and Shapes 14

Visibility 14

Damage 14

Event Consumption 15

Offscreen 15

Cursor 16

2.3. Colors 16

Chapter 3 Input

3.1. Input Events

3.2. Submitting Events for Distribution

3.3. Interests: Event Selection

Changing and Reusing Interests

Inquiring for Current Interests

3.4. Event Distribution

Receiving Events

Event Matching

Name and **Action** Matching

Canvas Matching

Process Matching

Processing after an Interest Match

Order of Interest Matching

Interest Lists

Multiple Matches

3.5. Special Event Types

Actions for Enter and Exit Events

3.6. Input Synchronization

3.7. Event Logging

3.8. Example

Chapter 4 Extensibility through POSTSCRIPT Language Files

4.1. Server Initialization

4.2. File Organization

POSTSCRIPT language Files Loaded at Initialization

`init.ps`

`systoklst.ps`

`colors.ps`

`cursor.ps`

`statusdict.ps`

`icon.ps`

`util.ps`

litemenu.ps 32

demomenu.ps 32

litewin.ps 32

liteUI.ps 32

POSTSCRIPT language Files for User Settings 32

user.ps 32

startup.ps 32

Other POSTSCRIPT language Files 33

compat.ps 33

liteitem.ps 33

litetext.ps 33

debug.ps 33

eventlog.ps 33

journal.ps 33

repeat.ps 33

4.3. Some POSTSCRIPT language Extensions in the *.ps Files 33

Miscellaneous 33

4.4. User Interaction and Event Management 36

4.5. Rectangle Utilities 38

4.6. Graphics Utilities 39

4.7. CID Utilities 40

4.8. Text and Font Utilities 41

4.9. Journalling Utilities 42

Journalling Internal Variables 43

4.10. Constants 43

4.11. Key Mapping Utilities 43

4.12. Repeating Keys 44

4.13. Colors Definitions 45

4.14. Logging Events 45

UnloggedEvents 45

Chapter 5 The Extended Input System **49**

5.1. Building on NeWS Input Facilities 49

5.2. The LiteUI Interface ....

5.3. Keyboard Input ....

Keyboard Input: Simple ASCII Characters ....

Keyboard Input: Function Keys ....

Assigning Function Keys ....

Keyboard Editing and Cursor Control ....

5.4. Selection Overview and Data Structures ....

Selection Data Structures ....

Selections: Library Procedures ....

Selection Events ....

/SetSelectionAt ....

ExtendSelectionTo ....

DeSelect ....

ShelveSelection ....

SelectionRequest ....

5.5. Input Focus ....

Chapter 6 Classes ....

6.1. Packages and Classes ....

6.2. Introduction to Classes ....

6.3. Class 'Foo' ....

References ....

Chapter 7 Window and Menu Packages ....

7.1. Package Style ....

7.2. A Scrollbar Implementation ....

Description Format ....

7.3. Menu Methods ....

Polymorphic Menu Keys ....

7.4. Window Methods ....

7.5. An example: `lines` ....

7.6. Default User Interface ....

PointButton ....

AdjustButton 83

MenuButton 83

Modifying the User Interface 83

Chapter 8 Debugging **87**

8.1. Introduction 87

Contacting the Server 87

Starting a Debugging Session 88

8.2. The Debugging Environment 88

Multi-Process Debugging 88

8.3. Client Commands 88

8.4. User Commands 89

8.5. Miscellaneous Hints 93

Aliases 93

Use Multiple Debugging Connections 93

Chapter 9 C Client Interface **97**

9.1. How to Use CPS 97

The `.cps` File 98

The `.h` File 100

The `.c` File 100

Comments 101

9.2. Tags, Tagprint, Typedprint 101

Tags 101

Receiving Tagged Packets from NeWS 102

9.3. A Sample Tags Program 103

9.4. Tokens and Tokenization 105

9.5. The CPS Utilities 106

Chapter 10 A Complete Example: `roundclock` **109**

`roundclock.c` 109

`roundclock.cps` 111

Chapter 11 NeWS Type Extensions **117**

11.1. New Objects in NeWS
11.2. Objects as Dictionaries
11.3. Canvases as Dictionaries
11.4. Events as Dictionaries
11.5. Graphics States as Dictionaries
11.6. Processes as Dictionaries
11.7. Shapes as Dictionaries
11.8. Object Cleanup
Server Function
Object Management
Error Handling
Connection Management
Process Management
Killing An Application
Garbage Collection
11.9. NeWS Security

Chapter 12 NeWS Operator Extensions

Chapter 13 Omissions and Implementation Limits
13.1. Operator Omissions
13.2. Imaging Omissions
13.3. Implementation Limits
13.4. Autobind

Chapter 14 Byte Stream Format
14.1. Encoding
14.2. Object Tables
14.3. Magic Numbers
14.4. Examples

Chapter 15 Supporting NeWS From Other Languages
15.1. Contacting the Server
15.2. Communication with the Server

Chapter 16 Font Tools **167**

16.1. Cursor Fonts 167

A Standard Font 167

Representation 167

Format 168

Generating a Font 168

16.2. Building an Ordinary Font 170

Appendix A Using NeWS **173**

A.1. NeWS Environment Variables 173

Which Server Binary? 174

The Debugging Server Binary 174

A.2. Starting up NeWS 174

From outside `suntools` 174

From within `suntools` using `overview(1)` 174

Server Initialization 174

A.3. SunView1 Binary Compatibility with NeWS 174

Bugs in SunView1/NeWS Coexistence 175

Inconveniences 175

Screen Damage 176

Input Mismatches 176

NeWS on the Sun-3/110 176

A.4. Learning NeWS 177

Putting A Message in a Window 177

The `psh` Command 177

Running POSTSCRIPT language Programs 177

Using Journalling 177

Previewing POSTSCRIPT language Graphics 178

Talking Directly to the Server 178

A Sample Session 178

Connecting to Remote NeWS Servers 179

A.5. A Sample `psh` Program: `test.psh` 180

A.6. Dictionaries and the Server 183

Modifying the NeWS Server ....

`startup.ps` ....

`user.ps` ....

Notes on Modifications ....

Modifying Your ''Root'' Menu ....

Saving Keystrokes ....

Changing Defaults ....

Appendix B Class *LiteItem* ....

B.1. Class **Item** ....

B.2. Two Sample Items ....

Sample Items Test Program ....

B.3. Class **LabeledItem** ....

B.4. Subclasses of **LabeledItem** ....

B.5. **LabeledItem** Subclass Details ....

Appendix C NeWS Operators ....

C.1. NeWS Operators, Alphabetically ....

C.2. NeWS Operators, by Type ....

Appendix D NeWS Manual Pages ....

Index ....

Tables

Table 3-1 Boundary Crossing Events 25

Table 4-1 Standard NeWS Cursors 38

Table 5-1 Selection-Dict Keys 54
Table 5-2 System-defined Selection Attributes 54
Table 5-3 Request-dict Entries 54

Table 7-1 **LiteWindow** Instance Variables 78
Table 7-2 Window User Interface Button Usage 82

Table 9-1 CPS Argument Types 99

Table 12-1 Mouse Event Translation 139

Table 13-1 Omitted POSTSCRIPT language primitives 151
Table 13-2 Implementation Limits 152

Table 14-1 Token Values 159
Table 14-2 Meaning of Bytes in Encoding Example 159

Figures

Figure 1-1 Client – Server Interaction in NeWS 4

Figure 6-1 Relationship between Instances and Classes 64
Figure 6-2 Self and Super 64
Figure 6-3 POSTSCRIPT language use of Dictionaries as Objects 65
Figure 6-4 POSTSCRIPT language use of Dictionaries as Objects 66

Figure 9-1 Short Tags Specification 102
Figure 9-2 Long Tags Specification 102
Figure 9-3 A Server Side Tags Program 103
Figure 9-4 A Client Side Tags Program 104

Figure B-1 Two Instances of Class 'SampleToggle' 192
Figure B-2 An Instance of Class 'SampleSlider' 193
Figure B-3 The Sample Test Program 194
Figure B-4 Use of the Sample Test Program 195
Figure B-5 Use of the Sample Test Program — Moving the Slider 196
Figure B-6 Subclasses of **LabeledItems** 199
Figure B-7 Typical Item Usage 200

Preface

This manual is a combination guide and reference to NeWS.

Prerequisite Documents

All of this manual but the *Introduction* assumes knowledge of the material covered in Adobe's *PostScript Language Reference Manual*, published by Addison-Wesley. If you are unfamiliar with the POSTSCRIPT language, you should also consider the companion book *PostScript Language Tutorial and Cookbook* required reading.

Companion Documents

The *NeWS Technical Overview* is a useful introduction to the concepts and benefits of NeWS.

Where to Start

General help may be found in Appendix A, *Using NeWS*, and you can work through the examples in the *PostScript Language Tutorial and Cookbook* using NeWS's `psview` command.

Structure of the Manual

The manual is organized as follows:

- An introduction to the design and goals of NeWS.
- An overview of the NeWS extensions.
- Further information about the extensions for handling input and events.
- An overview of the POSTSCRIPT language files that implement additional server functionality and *packages*.
- Descriptions of some of the packages implemented in these POSTSCRIPT language files:
  - an extended input system
  - a complete classing mechanism
  - example window and menu packages based on this class mechanism
  - a debugging facility
- A description of the CPS program, used to construct C client interfaces to NeWS.
- A complete example client program.

- Reference chapters for the NeWS types and operators.
- A section detailing the areas in which the current POSTSCRIPT language implementation is incomplete.
- A reference section for the format of the byte stream communication between the clients and the server, detailing the data compression techn available.
- Information on font support in NeWS.
- An appendix on how to start up NeWS, with some advice on programmi NeWS and customizing the NeWS server.
- An appendix with a complete listing of all the NeWS operators, sorted al betically and by class.
- Details of another class package, *LiteItem*, used in the `itemdemo` der program.
- An appendix containing manual pages for NeWS.

Font Usage

The NeWS Manual includes code and procedures from two different languag and the POSTSCRIPT language. We have used fonts to clarify which langua used. This differs from other Sun manuals:

`bold listing font`
: This font indicates things that you should type at your v tation.

`listing font`
: This font indicates literal values such as file names and put displayed by the computer. It also indicates use of language: it is used in C program listings and C proced names. CPS routines and code fragments such as `ps_open_PostScript()` are printed in this font.

sans serif font
: This font is used for *POSTSCRIPT* program listings, type code fragments such as 300 200 createcanvas map vas to distinguish them from C code. It is also used in definition of NeWS functions (primarily in Chapter 12, *N Operator Extensions*).

bold font
: Unfortunately, sans serif fonts look poor in the middl normal text. So, as well as indicating cautions and war bold font is used to indicate all NeWS names, such as **clipcanvas**, when they appear in paragraphs or the inde

italic font
: This font is used as a place holder for words, numbers, expressions that you define, for example parameters to mands, and operands of POSTSCRIPT language operato Italics are also used in the conventional manner to emp important words and phrases.

1

Introduction

Introduction 3

1.1. The Design 3
1.2. Imaging 5
1.3. Canvases 5
1.4. User Interaction — Input 6
1.5. Client Interface 6
1.6. Background and Goals 7
Device Independence 7
Portability 8
Flexibility 8
Distribution 9
Graphics 9
References 9

1

Introduction

NeWS is a distributed, extensible window system that takes a long-term approach to the development of user interface and display technology. It is not an attempt to codify and build on existing systems; it is rather an attempt to step up to a new level of technology. The unique feature of NeWS is the ubiquitous use of an extension mechanism. The extensibility of the system is the key to integrating windows efficiently into a distributed environment. Performance is enhanced by close interaction between clients and their server, communication is speeded up by application-specific data compression, semantic issues are reduced by a central authority, and user interface issues are easier to address.

1.1. The Design

NeWS is based on a novel sort of interprocess communication. Interprocess communication is usually accomplished by sending messages from one process to another via some communication medium. Messages are usually streams of commands and parameters. One can view these streams of commands as programs in a very simple language. What happens if this language is extended to become Turing-equivalent? Programs no longer communicate by sending messages, they communicate by sending programs that are elaborated by the receiver. This has interesting effects on data compression, performance, and flexibility.

The POSTSCRIPT programming language defined by John Warnock and Charles Geschke at Adobe Systems is used in just this way.[A] What Warnock and Geschke were trying to do was communicate with a printer. They transmit programs in the POSTSCRIPT language to the printer, which are elaborated by a processor in the printer, and this elaboration causes an image to appear on the page. The ability to define a function allows the extension and alteration of the capabilities of the printer.

This idea has powerful implications within the context of window systems: it provides a graceful way to make the system much more flexible and it provides some interesting solutions to performance and synchronization problems. For example, if you want to draw a grid, you don't have to transmit a large set of lines to the window system, you just send a program containing the appropriate iteration. Downloading programs in an extension language is not just a nice feature that has been tacked on; it is an integral part of the window system.

NeWS extensions conform to the form of POSTSCRIPT primitives. The POSTSCRIPT language is clean and simple, it has a well-designed graphics model,

and it is compatible with many of the printers available today.

NeWS is a single process, which acts as a network server and contains a POSTSCRIPT language interpreter[1]. Within this server process is a collectior lightweight processes that execute POSTSCRIPT programs. A lightweight pr is unlike a typical UNIX process in that it shares a data space with other ligh weight processes. Consequently, process creation has very little overhead ai characterized by great rapidity.

Client programs talk to NeWS through byte streams. Each of these streams g erally has a lightweight process within the NeWS server that executes the stre

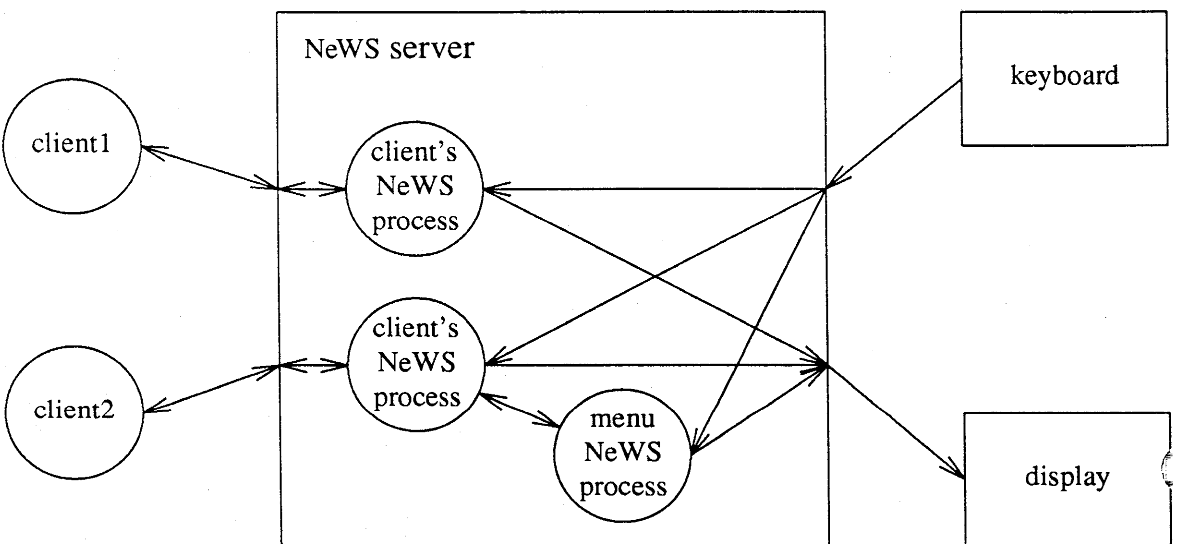


Figure 1-1 *Client – Server Interaction in NeWS*

Messages pass between client processes, which exist somewhere out on the work, and NeWS processes, which exist within the NeWS server. These proce can perform operations on the display and receive events from the keyboarc the mouse. They can talk to other NeWS processes, which may, for example implement menu packages.

NeWS centers around the POSTSCRIPT language. All that is provided by NeW: set of mechanisms; policies are implemented as POSTSCRIPT procedures. F example, NeWS has no window placement policy. It has mechanisms for cre windows and placing them on the screen, given coordinates for the window choice of those coordinates is up to some POSTSCRIPT procedure.

What is usually thought of as the *user interface* of a window system is expl outside the design of this window system. User interface includes such thir how menu title bars are drawn and whether or not the user can stretch a win by clicking the left button in the upper right hand corner of the window outl All these issues are addressed by implementing appropriate procedures in tł

[1] NeWS was conceived and created wholly by Sun Microsystems.

POSTSCRIPT language.

The rest of this section presents NeWS in four parts: the imaging model, window management, user interaction, and the client interface. The imaging model refers to the capabilities of the graphics system — the manipulation of the contents of a window. Window management refers to the manipulation of windows as objects themselves. User interaction refers to the way a user at a workstation interacts with the window system (e.g., how keystrokes and mouse actions are handled). The client interface defines the way in which client programs interact with the window system (e.g., how programs make requests to the window system).

1.2. Imaging

Imaging in NeWS is based on the stencil/paint model, essentially as it appears in Cedar/Graphics[B] and the POSTSCRIPT language. A stencil is an outline specified by an infinitely thin boundary composed of spline curves in a non-integer coordinate space. Paint is some pure color or texture — even another image — that may be applied to the drawing surface. Paint is always passed through a stencil before being applied to the drawing surface, just like silkscreening. This is the *total* model. Lines and characters can be defined using stencils. Lines are done as long, narrow stencils.

One of the attractive characteristics of this imaging model is its very abstract nature. For example, the definition of a font allows many implementations: as bitmaps, as pen strokes, or as spline outlines. No commitment is made about exactly which pixels are affected, or even that there are pixels at all. The extension of the system to deal with anti-aliasing does not affect the interface. The use of a very abstract imaging model provides a very high degree of device independence.

The specification of this model is simple and elegant, but the way in which its various features can be combined leads to a tricky implementation. For example, the mechanism for specifying a stencil allows straight lines, arcs, and higher-order curves to be a part of its boundary. Stencils can be used both for clipping and for filling. This implies that it must be possible to compute the intersection of curved boundaries.

NeWS implements curves with conic splines.[C] Curves form *paths*, or shapes, and NeWS has a set of algorithms for manipulating these paths.[D] These algorithms have been assembled into a library that supports the stencil/paint model. The NeWS server is implemented as a language interpreter that knows nothing about imaging, but calls routines in this library to perform all imaging operations.

1.3. Canvases

NeWS's basic drawing surface is a *canvas*. This non-standard term was picked to avoid the semantic confusion that surrounds the word 'window.' A canvas is just a surface on which an image may be drawn. The surface may be either opaque or transparent, and can have any shape. Canvases are laid out in two-and-a-half dimensions on a display surface; in other words, they can overlap. The actual implementation of canvases depends heavily on the graphics package described in the previous section. A canvas may keep a portion of its image off-screen in order to facilitate quick refresh of its image when uncovered. In addition, a canvas may be drawn to while not displayed and then *mapped* to the display that renders it visible; this is a method for double-buffering.

Canvases are cheap and easy to create. Menus, windows, and pop-up m
are all based on canvases. NeWS has extensions in the form of primitives
and manipulate canvases. All POSTSCRIPT graphics operations are perform
some canvas.

1.4. User Interaction — Input

Each possible input action is an *event*. Events are a general notion that incl
buttons going down and up (buttons may be on keyboards, mice, tablets, or
ever else) and locator motion. They are implemented as messages between
processes.

Events are distinguished by where they occur, what happened, and to what.
objects spoken about here are usually physical; they are the things that a pe
can manipulate. An example of an event is the [E] key going down while t
mouse is over canvas *x*. This might trigger the transmission of the ASCII c
for E to the process that created the canvas. The bindings between events a
actions are very loose and easy to change.

The actions to be executed when an event occurs can be specified in a gene
way, via the POSTSCRIPT language. The striking of the [E] key sends a me
to a NeWS process that is responsible for deciding what to do with it. The p
can do something as simple as sending the message to a UNIX process, or a
complicated as inserting the message into a locally maintained document.

POSTSCRIPT language procedures control much more than just the interpret
of keystrokes. They can be involved in cursor tracking, constructing borde
around windows, doing window layout, and implementing menus. Th p
cedures strongly resemble the Bell Labs *squeak* language, with lightw t
processes replacing concurrency compilation.[E]

1.5. Client Interface

A client program exists in two parts: one part is written in the POSTSCRIPT
language and lives inside NeWS, and one part lives outside NeWS and talks t
through a byte stream. This leads to a number of levels at which the client
face can be viewed.

- At the lowest level, the programmer writes POSTSCRIPT language prog
and deals with an entirely POSTSCRIPT language universe. Menu pack
and window layout policies are examples of objects that will usually b
implemented this way.

  At this level, NeWS provides conventions that define an object-oriented
face to windows, menus, selections, and so on. Objects inherit a defa
of behaviors, but these can be overridden selectively.

- One step above that, the programmer writes programs in C, or some o
language, that generate POSTSCRIPT language programs. The progran
explicitly aware of the existence of the POSTSCRIPT language. NeWS e
tors of other window systems are generally implemented this way.

  At this level, NeWS provides a C pre-processor that allows C programi
cross the language boundary to the POSTSCRIPT language easily. In e
they can write C procedures with POSTSCRIPT language bodies. Anal
tools for other languages are possible.

- At the highest level, the existence of the POSTSCRIPT language and message passing is completely hidden by an interface veneer that someone else has constructed using the second-level facilities. NeWS appears as a set of routines that are called in the normal way.

As with the user interface, NeWS defines no details of the programmer's interface and permits it to be specified by POSTSCRIPT programs. The programmer's interface may even be created on a per-application basis, by writing POSTSCRIPT programs.

1.6. Background and Goals

In some respects, NeWS represents a major break with the technology used in current window systems. In other respects, it is very similar to some existing systems. This section examines these differences and similarities, relating them to the overall design goals of NeWS.

Device Independence

Most current window systems, including *SunWindows*,[F] MIT's *X*,[G] and Carnegie-Mellon's *Andrew*,[H] are based on the RasterOp and pixel coordinate imaging model. As graphics hardware evolves to provide better performance, this model becomes less appropriate:

- The model preempts the use of advanced transformation and rendering hardware by insisting clients break down their graphical operations to such a low level that the hardware is useless. Current window systems mean that powerful hardware assists only those clients which take special measures to use it.
- The model ensures that clients are aware of the actual size of individual pixels on the display, since the only coordinate system they have is that provided by the hardware. Until recently, this has not been a severe problem because the pixels on displays in use have all been within 30% or so of 80 dots to the inch. But displays up to 300 dots to the inch are already becoming available, and resolutions may go much higher.
- The model doesn't extend in a clean and useful way to color. Boolean combination functions between color pixel values don't make much sense. For instance, one often draws transient rubber band lines by XORing them with the image. XORing color map indices can lead to some pyrotechnic effects. Furthermore, the model exposes the differences among the three common ways that color is represented in display devices: 1-bit black and white (constant small set of colors), 8-bit color with a colormap (variable small set of colors), and 24-bit color (all possible colors available everywhere). Clients have to invoke different operations for each display type.

NeWS' more abstract imaging model allows advanced transformation and rendering hardware to assist all clients; clients do not have to determine if it is available and take special measures to use it. NeWS clients need not be concerned with the hardware coordinate system; they define their own.

NeWS clients also need not be concerned with any dependencies on a particular color model or on color hardware. NeWS allows clients to specify colors as red–green–blue values or as hue–saturation–brightness and will make its best

efforts to display the correct colors on the screen. On full-color displays, displays the exact color. On color-mapped displays, NeWS selects the closest color table entry to the requested color. On monochrome and gray-scale displays, NeWS will use dithering and half-toning techniques as necessary. Ir way, clients expend minimal effort to get usable output on a wide variety of displays.

A similar problem of device-dependence exists on the input side of current v dow systems. They implement a fixed set of devices, and the client has to b aware of the set. In many cases, the client has to determine which devices a ally are present and load appropriate keyboard mapping tables. In NeWS, arl trary transformations of input events may be programmed in the POSTSCRIP language, so each client can be presented with an appropriate set of input de ices, and new and unanticipated devices can be accommodated.

Portability

Even if current window systems did not expose so much of the underlying display hardware, many would not be portable between different machines. tems that are implemented largely as part of the operating system kernel, su *SunWindows*, demonstrate this problem.

NeWS, like *X* and *Andrew*, is a user-level server process. Both these predece have been ported with relatively little effort to a range of workstations. NeW like *Andrew* in that it has been designed from the start to be ported. A wind system that is available over a wide range of workstations and displays is be ing a necessity for the UNIX marketplace.

Flexibility

Some current window systems, *Andrew* and the Macintosh[I] are examples, s to impose a single consistent user interface style on their clients. This has g productivity benefits, both for the user and the programmer. Users can lear applications easily, because they behave the same as the last application lea Programmers can inherit much of the user interface of a new application fro "canned" application like MacApp.

Other current window systems, *X* and *Smalltalk*[J] are examples, observe that wide range of interface styles are already in use and that a specialized interf can make an experienced user more effective than any single consistent inte They set an explicit design goal of avoiding specifying a user interface styl other window systems the style is determined for the most part by libraries into each application, and is very difficult either to change or to make consi

NeWS attempts to address both aspects of this dilemma. The server does no depend on any details of the user interface; it is written in the POSTSCRIPT language and thus are easily changed. The way the lightweight NeWS proce share their name spaces encourages clients to share user interface compone such as menus, scrollbars, and so on. These shared components are typicall inherited from the global POSTSCRIPT language environment, and are thus sistent across the range of clients. They can be replaced for all applications changing the global environment. The effect is that unless clients explicitl override the default behavior, they are consistent in their user interface.

Distribution

Both *Andrew* and *X* have demonstrated that windows can be presented as a network service, available to clients anywhere, and can provide adequate performance for most tasks. In a highly networked environment, a window system that restricts clients to running on the machine with the display is unnatural.

Like these systems, NeWS is a network window server whose clients can be anywhere. However, it takes the provision of adequate performance a step further. The ability to write the parts of a user interface needing instantaneous response, such as rubber-band lines, in the POSTSCRIPT language and have them executed by the server without client intervention solves one of the critical performance problems of current systems. The ability to down-load interpreters for special, application-specific protocols allows better use to be made of the limited network bandwidth.

Graphics

A window system that provides only RasterOp, lines, and simple text makes the construction of graphically interesting interfaces difficult. The Macintosh, for example, is a system that has a richer graphics model and as a result has a flair for more interesting interfaces at the cost of a somewhat more complex application programmer interface. NeWS's use of the higher-level stencil/paint imaging model provides advanced imaging capabilities without increasing the apparent complexity.

References

A. Adobe Systems, *POSTSCRIPT Language Reference Manual*, Addison-Wesley, July, 1985.

B. John Warnock and Douglas Wyatt, ''A Device Independent Graphics Imaging Model for Use with Raster Devices,'' *Computer Graphics*, vol. 16, no. 3, July, 1982.

C. Vaughan Pratt ''Techniques for Conic Splines,'' *SIGGRAPH Proceedings*, July, 1985.

D. James Gosling, ''If the Earth is Round, Why is the Sun Square,'' *Usenix Workshop on Computer Graphics*, Monterey CA, December, 1985.

E. Luca Cardelli and Rob Pike, ''Squeak: A Language for Communicating with Mice,'' *SIGGRAPH Proceedings*, July, 1985.

F. *Programmer's Reference Manual for SunWindows*, Sun Microsystems, April, 1985.

G. James Gettys, ''Problems Implementing Window Systems in Unix,'' *Usenix Proceedings*, Denver, CO, January, 1986.

H. David Rosenthal and James Gosling, ''A Window Manager for Bitmapped Displays and Unix,'' in *Methodology of Window Managers*, ed. F. R. A. Hopgood et al., North Holland, 1986.

I. C. Espinosa and C. Rose, *QuickDraw: A Programmer's Guide*, Apple Computer, March, 1983.

J. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, May, 1983.

2

NeWS Extension Overview

NeWS Extension Overview **13**

2.1. The Lightweight Process Mechanism 13

2.2. Canvases and Shapes 14

Visibility 14

Damage 14

Event Consumption 15

Offscreen 15

Cursor 16

2.3. Colors 16

2

NeWS Extension Overview

NeWS implements a number of extensions to the POSTSCRIPT language (as specified by Adobe in the *PostScript Language Reference Manual*) which are specific to NeWS for the purposes of interactive behavior. These extensions include new types and new operators. The POSTSCRIPT language was initially designed for driving printers. As a part of NeWS, the POSTSCRIPT language interpreter has to deal with multiple asynchronous clients, interaction, displays, keyboards, and locators. This chapter is an overview of the NeWS extensions in the areas of lightweight processes, *canvases*, colors and cursors. For a full description, see Chapter 11, *NeWS Type Extensions* and Chapter 12, *NeWS Operator Extensions*.

Many application programmers will not be concerned with most of the primitives described here. They will use instead the packages that have been defined using these primitives to support windows, menus, and the like. These packages of POSTSCRIPT language code are described starting in Chapter 4, *Extensibility through POSTSCRIPT Language Files*.

2.1. The Lightweight Process Mechanism

The NeWS server maintains a set of simultaneously executing lightweight processes. Each process is an individual thread of control with its own graphics context, dictionary stack, execution stack, and operand stack. These 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. This thread can fork new threads and form a group of NeWS processes. Such groups of processes are represented by process objects.

Processes can fork new processes, kill them, wait for them to die and obtain a return value, pause to allow other processes to run, suspend themselves and other processes, continue suspended processes, and examine the state of other processes by opening the process objects that represent them as dictionaries.

The lightweight process scheduling policy is *non-preemptive* (a process continues to execute until it blocks) and *serial* (only one process is active at a time). Processes block by executing file I/O requests, the **pause** or **suspendprocess** primitives, or **awaitevent**.

NOTE *The current scheduling policy might change to pre-emptive scheduling in the future. Thus, it is unwise to write POSTSCRIPT language code that relies on the behavior of a non-pre-emptive scheduling policy. POSTSCRIPT language code*

that accesses shared data structures should use monitors to protect tho d structures.

2.2. Canvases and Shapes

A *canvas* is a surface upon which are drawn using the POSTSCRIPT languag Each NeWS process has a canvas associated with it called the *current canva* Canvases exist in a hierarchy. At the root of a hierarchy is a device canvas device canvas is created as the result of the **createdevice** operator and repr the background, sometimes called the ''desktop.'' Additional canvas obje may be created with **newcanvas** calls.

A canvas can be repositioned in the list of its siblings with **canvastotop** an **vastobottom**. Its *x,y* offset relative to its parent may be set with **movecan** canvas may be inserted inserted above (**insertcanvasabove**) or below (**ins vasbelow**) another canvas.

Canvases need not be rectangular. Their shape may be set with **reshapeca** to be the region outlined by the current path. Each canvas also has associa with it a default transformation matrix. The **reshapecanvas** operator sets vas' default transformation matrix from the current matrix.

Visibility

If a canvas is to be visible on a display device, it and its ancestors must be *mapped*. Canvas mapping is controlled by the **Mapped** field of a canvas, is a boolean value. When a canvas is created, it is initially unmapped, so t value of its **Mapped** field will be **false**. Setting this field to **true** and **false** map and unmap the canvas from the display.

A canvas may be *transparent*: its image does not obscure any image draw underneath it by a parent or sibling. Any image drawn on a transparent ca drawn on its parent. A non-transparent canvas is referred to as being *opaq* Transparency is controlled by the **Transparent** field of a canvas, a boolea Transparent canvases are useful for defining areas that are sensitive to inp that do not interfere with drawing in other canvases.

Damage

A canvas is considered to have *damage* if all or part of its image does not There are several ways that damage can occur. A canvas may become da when, for example, another canvas is moved away from it, exposing the fi vas. The entire canvas is considered to be damaged when it is first mappe the screen (if it is not retained) or when it makes the transition from non-r to retained. The entire canvas is also considered to be damaged whenever reshaped.

All programs have to cope with canvas damage and must be able to recon the damaged part. An opaque canvas may be *retained;* that is, any portio canvas obscured by other canvases is saved in some offscreen area. Whe these obscured areas is exposed, the offscreen copy is simply moved onto screen. If the canvas were not retained, there would be no copy, and the c would be damaged. Retaining a canvas is purely a performance enhancen

Damage can be spread with **copyarea**, which copies a region on the canv one place to another. If part of the source is damaged, the correspond tion area becomes damaged.

Damage accumulates on a canvas until some process responds to it. Each opaque canvas has a record of its damaged regions. As more damage occurs, this record is enlarged. The **damagepath** operator sets the current path to an outline that encloses all the damaged parts of the canvas, *and clears the damage record.* The sequence of events followed to deal with damage repair is generally:

- Damage occurs on an opaque canvas.
- A **/Damaged** event is generated.
- A NeWS process receives the event.
- The NeWS process sends a message to the client program informing it that damage has occurred.
- The client program receives the message and sends a message back to initiate the repair.
- When the repair initiation message is received by a NeWS process, it executes **damagepath clipcanvas** for the opaque canvas that was damaged.
- The NeWS process may send back a description of the region damaged.
- The client program sends a NeWS program to the server that will redraw the damaged region (or it will draw the whole window and let the **clipcanvas** operation throw away irrelevant operations).
- The client program sends an end-of-repair message that executes **newpath clipcanvas**.

This multi-level handshake is used so that the client and server can proceed asynchronously and yet be properly synchronized when they deal with damage.

Event Consumption

Canvases also have a property that controls their behavior with respect to input events. The **EventsConsumed** field controls what happens to events that occur on this canvas. The **EventsConsumed** field can have one of three keyword values: **/AllEvents, /MatchedEvents,** or **/NoEvents**. If **EventsConsumed** equals **/AllEvents**, all events on this canvas are consumed by it. That is, no canvas behind this one will receive any events. If **EventsConsumed** equals **/MatchedEvents,** then events that match an interest on this canvas will be consumed, while non-matching events will be passed through to canvases behind this one[2]. Finally, if **EventsConsumed** is equal to **/NoEvents**, no events will be consumed, and all will be pass through to the canvases behind.

Offscreen

A NeWS program can maintain an offscreen image by use of a canvas that is retained, unmapped, and opaque. A process can draw in one of these canvases in the same way it draws on other canvases, the only exception being that the image will not appear on the screen. One way to have the image appear is to simply map the canvas onto the screen. Another way of having the image appear is to use the **imagecanvas** operator to copy the image from the offscreen canvas to an onscreen canvas.

---

[2] For an explanation of interests and events, see Section 3.3, *Interests: Event Selection* of Chapter 3, *Input.*

Cursor

Every canvas has a *cursor* image that is displayed whenever the mouse is over the canvas. This image is set with **setcanvascursor** and retrieved with **getcanvascursor**. A child canvas inherits its parent's cursor upon creation of the canvas. A cursor image is composed of a black *primary image* and a white *image*. These images are superimposed on their origins. These images are ing more than characters in a font. The *hot spot* of a cursor (the pixel coord to which the mouse is pointing) is the origin of the primary image's charact

The current location of the cursor in the current coordinate system is returne **currentcursorlocation**. Similarly **setcursorlocation** will move the cursor new location. We discourage use of **setcursorlocation** because it causes di certing cursor jumps.

2.3. Colors

The POSTSCRIPT language has a notion of color, which is implemented in th **sethsbcolor** and **setrgbcolor** primitives. NeWS extends this notion by addin color type. Objects of type **color** can be created with either the HSB or the color model. Color objects can also be compared with the **contrastswithcu** primitive.

3

Input

Input **19**

3.1. Input Events 19
3.2. Submitting Events for Distribution 20
3.3. Interests: Event Selection 20
Changing and Reusing Interests 21
Inquiring for Current Interests 21
3.4. Event Distribution 21
Receiving Events 21
Event Matching 21
Name and **Action** Matching 22
Canvas Matching 22
Process Matching 22
Processing after an Interest Match 22
Order of Interest Matching 23
Interest Lists 23
Multiple Matches 23
3.5. Special Event Types 24
Actions for Enter and Exit Events 24
3.6. Input Synchronization 25
3.7. Event Logging 26
3.8. Example 27

3

Input

Printers don't receive input from the user; conversely, window programs do much more than display graphics. For example, they usually change their display in response to outside events such as mouse clicks. Hence, input handling in NeWS is a major extension to the POSTSCRIPT language. This chapter describes the primitive data structures and operations for handling input in NeWS. These primitives provide a minimal user-input system. More sophisticated user-interfaces can be constructed entirely in the POSTSCRIPT language on top of these primitives (for example, the *Extended Input System* described in Chapter 5, *The Extended Input System)*.

3.1. Input Events

Input in NeWS is treated as a series of *events* that are received, translated, dispatched, and routed by the server to its clients. Events are structured objects. They contain a number of fields, which are accessed as though the event were a dictionary and the fields were keys in that dictionary. Most of these fields are mentioned as they become relevant in this chapter. The full definition of event fields can be found in Chapter 11, *NeWS Type Extensions*. The POSTSCRIPT language interface is structured in such a way that adding fields does not affect existing POSTSCRIPT programs.

Among the most interesting fields in an event are:

- An event **Name** (a POSTSCRIPT language object — generally a small number to represent a character, or a keyword such as **/LeftShift** or **/MouseDragged**, but any object is legal.
- An event **Action** (another POSTSCRIPT language object, with more variety in common types and semantics).
- A **TimeStamp**, which shows when the event happened.
- A **Canvas** and a coordinate pair (**XLocation**, **YLocation**), which give an event location in terms of the cursor position at the time of the event.

Many events are generated by the system to report user actions like mouse motion and key presses. Events can also be generated by NeWS processes, and submitted for processing just like system-generated events. The **createevent** primitive leaves a new event on the stack with null or 0 in all fields. The **copy** primitive can be used to copy the fields of one event into another *en masse*. This extension of the **copy** primitive is closely analogous to its usage with two dictionaries. (For the integrity of the input system, the event's **IsInterest** flag,

Serial number and **IsQueued** fields are not copied.)

3.2. Submitting Events for Distribution

Any event can be passed into the distribution mechanism to be received by a process whose interests it matches. Three primitives are used in this context

sendevent
takes an *event* off the stack. The event is sorted into the event queue an distributed as described below in Section 3.4, *Event Distribution*.

recallevent
takes an *event* off the stack and removes that event from the event queu This implies that the event must have been put in the queue by **sendeve** since no process will have a reference yet for system-generated events. also requires that the client must have saved some reference to the even given to **sendevent**, in order to pass the same event to **recallevent**. **recallevent** is useful for turning off a timer event that has been sent but yet delivered.

redistributeevent
takes an *event* (which should have been received by this process via **awaitevent**, or be a copy of such an event) and resumes the distributior cess right *after* the interest that resulted in the event's delivery to this p cess. (The following sections provide detailed discussions of interests.) This behavior is useful when a process receives an event via an *exclusi* interest (described below) and now needs to continue distribution (perh after modifications) as though the interest had not been exclusive.

redistributeevent does not return events to the event queue; **recalleve** will not work on a redistributed event. An interest that has been tested for a match against an event will not be tested again when the event is tributed; an interest will never match the same event twice. It is pointl pass an event that has never been distributed to **redistributeevent**; the will simply be discarded.

3.3. Interests: Event Selection

Processes indicate their interest in receiving events by constructing one or *ideal* events resembling what they want to receive and passing each ideal e as an argument to **expressinterest**. In other words, to get a particular kind event, create an event like it and express interest in that event. An event th been used as an argument to **expressinterest** in this way is called an *interes* When real events are generated and distributed, they are are matched to inte on the **Name**, **Action**, **Canvas**, and **Process** fields. Non-specific (''wild-ca matches may be specified, as described under Section 3.4.2, *Event Matchin* below. Two other event fields that affect the matching behavior of an inter its **Exclusivity** and its **Priority**; their effects are described under Section 3. *Order of Interest Matching* below.

An expression of interest may be canceled by **revokeinterest**. When intere expressed in an event, its **IsInterest** field is set to true. **IsInterest** is false f events if:

- they have never been passed to **expressinterest**,

- they have subsequently been passed to **revokeinterest**,
- the process that expressed the interest has terminated, or
- the canvas on which the interest was expressed is destroyed.

Changing and Reusing Interests

The **Name** and **Action** of an interest may be changed while it is active, and the change will be reflected in the next match attempted against that interest. However, changes to other fields in the interest will *not* be recognized and should not be attempted. Rather, the interest should be revoked, the modifications performed, and then the interest should be expressed again.

If an interest is used as the argument to a second invocation of **expressinterest** before it has been revoked or otherwise inactivated, the second expression of interest overrides the first. If the same process expresses interest in the same event twice in succession, the second expression is ignored.

Inquiring for Current Interests

The set of interests which are currently active for any canvas or process may be retrieved as an array of events, by treating the canvas or process as a dictionary, and getting the value associated with the **Interests** key. This mechanism is described fully in Chapter 11, *NeWS Type Extensions*. Similarly, the **global interest list** (the set of interests which have been expressed with **null** in their **Canvas**) is returned by the operator **globalinterestlist**. The result is an array of events, ordered on priority (highest first).

3.4. Event Distribution

Input events enter the system as they are generated by the NeWS server, or when a process executes **sendevent** or **redistributeevent**. Events generated by the server are stamped with the time of their creation; other events have whatever **TimeStamp** is left by the process that provides them. (A process can use the **currenttime** and **lasteventtime** primitives to generate a value for the **TimeStamp** field.) In any case, newly received events are sorted into a single event queue according to their **TimeStamp** values.

Events are removed from the head of the event queue one at a time as the server schedules processes to be run. No event will be distributed before the time indicated in its **TimeStamp**. Copies of the event are distributed to all processes whose interests it matches and each of those processes is given a chance to run before the next event is taken from the queue.

Receiving Events

A process gets its next input event by executing **awaitevent**. If no event has been distributed to it, the process will block. If a distributed event is waiting, **awaitevent** will return immediately, with the new event on the top of the operand stack.

Event Matching

Matching between a real event and an interest is defined as follows:

Name and **Action** Matching

- The **Name** and **Action** fields are treated the same.
- Null in an interest field matches anything in the corresponding field of real event.
- A simple object (boolean, keyword, or number) in the interest matches same value.
- An array or a dictionary in the interest specifies a class of values the re event may match. A real event value matches if it is any of the elemer the array, or keys in the dictionary.

Canvas Matching

A null **Canvas** in the interest matches events not directed to a specific canv This includes keystrokes and mouse-button/motion events. A non-null can the interest will match events occurring when the cursor is within that canv other events with the canvas field set to that canvas (e.g., **/Damaged** events

Process Matching

The **Process** field of an interest is set by **expressinterest** (to the process ex ing the interest). Normally, events being distributed have null in their **Pro** fields and will be matched against interests without restriction. If an event specific process in its **Process** field, the event will only match interests that been expressed by that process. (It must still match the interest on **Name**, **Action**, and **Canvas**.)

If all of these conditions are met, the event matches the interest.

Processing after an Interest Match

When an event matches an interest, a copy of the event is generated. I copy, the **Interest** and **Canvas** fields are set to the interest and canvas mat If the **Name** and/or **Action** values matched a key in a dictionary in the corresponding field in the interest, one of two things will happen:

1. If the value in the dictionary corresponding to the matching key is not cutable, then that value is stored in the **Name** or **Action** field in the ev
2. If the dictionary value *is* executable, then the value in the correspondi field of the event is not modified; instead, the executable object from t tionary is queued for execution in the receiving process immediately a the event is returned by **awaitevent**.

   If both the **Name** and **Action** fields of the event have such executable matches, the **Name** is executed first, then the **Action**.

Then the copy of the event is placed on a private queue for the process tha expressed the interest; if that process was blocked in **awaitevent**, it is mad runnable. The original event then may be matched against further interest

[3] The null-canvas option of an interest is not logically necessary; the same effect can be achieved b interest expressed on an overlay canvas for the root window. This overlay style has an additional bene allows recursive window managers. The null option has been retained for coding convenience among willing to forego the generality.

Order of Interest Matching

An event that is being distributed may potentially match more than one interest. This section describes which interests will be satisfied by the event. The order in which interests are considered for a match during the distribution of a real event is determined by the *interest list* each belongs to and by their order within those lists.

Interest Lists

Each interest is contained in one interest list. There is an interest list for each canvas; it holds all the interests that have been expressed on that canvas. There is also one *global interest list*, which contains all the interests expressed with a null **Canvas** field. An interest will never appear in more than one interest list, and any interest list may be empty.

When an event is being distributed, its **Canvas** field is checked, and if it is non-null, the event is matched only against interests on that canvas' interest list. If the real event's **Canvas** is null, the real event is first matched against interests on the global list. If none matches, it is matched against interests on the lists of canvases that contain the event's location. Canvas-specific interest lists are taken in leaf-to-root order in the canvas tree. That is, the interest-list of the front-most canvas is considered first, then the interest-list of its parent, then *its* parent, etc.

Within each interest list, the interests are ordered on their **Priority** field; higher numeric values come first. Among interests on the same list with the same **Priority**, the last-expressed interest is tested first.

Multiple Matches

The sequence of testing against interests continues until it is stopped by one of the following conditions:

1. Any interest may have its **Exclusivity** field set true; if so, an event that matches that interest will not be considered against any further interests.

2. A canvas may absorb events, so that they are not tested against interests on any canvas behind it. This is controlled by the canvas' **EventsConsumed** field.

   - If **EventsConsumed** is set to **/AllEvents**, no event that hits the canvas will be considered against the interest list of any canvas that lies behind it.

   - If **EventsConsumed** is **/MatchedEvents**, events that match an interest on that canvas' list will be stopped, but others will pass through.

   - If **EventsConsumed** is **/NoEvents**, events will be considered against interests on canvases farther back, regardless of whether they matched an interest on this canvas.

In these terms, the global interest list acts as if it were a list on a canvas that consumes **MatchedEvents**; events that match an interest on the global list don't get through to any canvas-specific list, unless they are explicitly redistributed.

3.5. Special Event Types

NeWS generates a number of different input events. Keystrokes generall numeric values in their **Name**, but most others are identified by a keyword **Name**. The most important event types are described here.

- **/Damaged**: Damage events are generated for a canvas whenever it is da aged. By the time a process repairs the damage, several events may ha accumulated. The total damage is accessible with **damagepath**. The **Action** for a damage event is null, and the **Canvas** field identifies the affected canvas.
- **/EnterEvent**, **/ExitEvent**: When the cursor is moved across a border between canvases, multiple events are generated. In each event, the Na either **/EnterEvent** or **/ExitEvent**, depending on the direction of the cr ing. Details of the **Action** are described in the next section.
- **/MouseDragged**, **/LeftMouseButton**, **/MiddleMouseButton**, **/RightMouseButton**: Manipulation of the mouse generates events with these names. If the mouse moves, the event **Name** is **/MouseDragged** the **Action** is null. If a mouse button is pressed or released, the **Name** identifies which button is affected and the **Action** is one of the keywor **/DownTransition** or **/UpTransition**.
- Timer events: There are no special timer events in NeWS; rather, the gua tee that no event will be delivered from the event queue before the time **TimeStamp** means that any event can be used to generate another even some time in the future. The example program at the end of this chapt illustrates a timer event.

  There is no requirement that a process send a timer event to itself; it ca as easily send a delayed message to another process, or broadcast one, changing the **Process** field in the event passed to **sendevent**.

Actions for Enter and Exit Events

Window-crossing events are generated whenever the cursor crosses the bou between two canvases. Each such event is directed to a particular canvas (identified in the **Canvas** field of the event) and each specifies how the curs moved with respect to that canvas.

The **Name** field of the event is either **/EnterEvent** or **/ExitEvent**. The **Ac** field of the event is 0, 1, or 2. The definitions below ease the explanation c these values mean.

Let us say that the frontmost canvas under the cursor *directly contains* the Then a canvas is *directly affected* by a crossing if it directly contains the cu either before or after the crossing. (If the same window directly contains tl sor both before and after an event, there is no crossing.)

A canvas may also be *indirectly affected*. This happens when the canvas is directly affected, but the cursor crosses into or out of the canvas' subtree. is, a canvas is indirectly affected if it is an ancestor of either the canvas tha directly contains the cursor before the crossing or the canvas that directly c tains the cursor after the crossing, but not both. (If a canvas is ancestor canvases that directly contain the cursor both before and after a crossi

affected.)

The following table explains the six combinations of **Name** and **Action** for crossing events.

Table 3-1 *Boundary Crossing Events*

Name	*Action*	*Explanation*
/EnterEvent	0	The canvas now *directly* contains the cursor; the previous direct container *was not* a descendant of this canvas.
	1	The canvas now *directly* contains the cursor; the previous direct container *was* a descendant of this canvas.
	2	The canvas now *indirectly* contains the cursor; the previous direct container *was not* a descendant of this canvas.
/ExitEvent		
	0	The canvas used to *directly* contain the cursor; the new direct container *is not* a descendant of this canvas.
	1	The canvas used to *directly* contain the cursor; the new direct container *is* a descendant of this canvas.
	2	The canvas used to *indirectly* contain the cursor; the new direct container *is not* a descendant of this canvas.

A crossing event (either **/EnterEvent** or **/ExitEvent**) is generated for every affected canvas, although there is no requirement that such events match any interest.

3.6. Input Synchronization

Processing of input events is synchronized at the NeWS process level inside the NeWS server. This means that all events are distributed from a single queue, ordered by the time of occurrence of the event, and that when an event is taken from the head of the queue, all processes to which it is delivered are given a chance to run before the next event is taken from the queue. When an event is passed to **redistributeevent**, the event at the head of the event queue is not distributed until processes that receive the event in its redistribution have had a chance to process it. No event will be distributed before the time indicated in its **TimeStamp**.

In some cases, a stricter guarantee of synchronization than this is required. For instance, suppose one process sees a mouse button go down and forks a new process to display and handle the menu until the corresponding button-up. The new process must be given a chance to express its interest before the button-up is

distributed, even if the user releases the button immediately. In general ever processing of one event may affect the distribution policy, distribution the next event must be delayed until the policy change has been completed. is done with the **blockinputqueue** primitive.

Execution of **blockinputqueue** prevents processing of any further events fr the event queue until a corresponding **unblockinputqueue** is executed, or a timeout has expired. The **blockinputqueue** primitive takes a numeric argur for the timeout; this is the fraction of a minute to wait before breaking the lc This argument may also be null, in which case the default value is used (currently 0.0083333 == .5 second). Block/unblock pairs may nest; the que not released until the outermost unblock. When nested invocations of **blockinputqueue** are in effect, there is one timeout (the latest of the set ass ated with current blocks).

Distribution of events returned to the system via **redistributeevent** is not affected by **blockinputqueue**, since those events are never returned to the e queue.

3.7. Event Logging

As an aid to developing NeWS applications, there is a facility for monitoring distribution of events by the NeWS server. A process may be designated as t *event-logger* by a call to **seteventlogger**. This process will be given a copy every event as it is distributed by the NeWS server. This includes events as t are taken for distribution from the input queue, and also events handed to **r tributeevent**. After having expressed some interest, the process should loo doing an **awaitevent**.

The interest is required so that **awaitevent** doesn't get a syntax error; it is not considered in the event-logging process. Therefore, it should not match any events; if it does, the process will receive two copies of those events. A -1 in the **Name** or **Action** is unlikely to match an event; an interest expressed on an unmapped canvas will also never match.

That **awaitevent** will return a copy of each event as it is distributed. The p may then do whatever it wishes with its copy; for example, it might print interesting fields in a window or a file. The file `eventlog.ps` described Chapter 4, *Extensibility through POSTSCRIPT Language Files* provides suck matted display of events.

The current event-logging process, or **null** if there is none. is returned by a to **geteventlogger**. Event-logging may be turned off by passing **null** as the ment to **seteventlogger**.

3.8. Example

The following short program illustrates many of the features of the NeWS input system described in this chapter. It prints clock ticks on its standard output for 15 seconds; then it prints a final message and goes away.

```
/clock {                                        % line 1
 {
    /d 5 dict dup begin
        /Tick /Tock def
        /Tock /Tick def                         % line 5
        /Pumpkin {
            (Pumpkin time....\n) print
            exit
        } def
    end def                                     % line 10
    /e1 createevent def
    e1 /Name d put
    e1 expressinterest
    /e2 e1 createevent copy def
    e2 begin                                    % line 15
        /Name /Pumpkin def
        /TimeStamp currenttime .25 add def
    end
    e2 sendevent
    /e3 e1 createevent copy def                 % line 20
    e3 dup begin
        /Name /Tock def
        /TimeStamp currenttime def
    end {
        dup begin                               % line 25
            /TimeStamp TimeStamp .016667 add def
        end sendevent
        awaitevent dup begin
            Name (     ) cvs print
            (...\n) print                       % line 30
        end
    } loop
  } fork
} def
```

In lines 3 – 10, the dictionary 'd' is defined to have three entries: '/Tick' and '/Tock' are defined to each other, and '/Pumpkin' is defined to be a small procedure that prints a message and exits.

This dictionary is then assigned to the **/Name** field of the event 'e1,' and 'e1' is passed to **expressinterest** (lines 12 and 13). This defines a class of events the 'clock' process will accept and, incidentally, specifies some processing to be done as the events are received. The events that 'e1' will match fit the following criteria:

- The **Name** must be one of '/Tick,' '/Tock,' or '/Pumpkin' (keys in the dictionary 'd' in the **Name**).

- Any **Action** is valid (null in the **Action**).
- The location of the event doesn't matter, but events directed to a specif canvas will not match (null in the **Canvas**).

Because the **Name** in the interest is a dictionary, special processing is invol matching events. If the event **Name** is '/Tick' or '/Tock,' it will be translat the other as the event is matched. (Non-executable values in the dictionary replace the value in event field.) If the event **Name** is '/Pumpkin,' it will r changed; rather, the value in 'd' of the procedure defined as '/Pumpkin' wi executed as the **awaitevent** at line 28 returns. (Executable values in the dic ary are queued for execution without changing the field in the event.)

Line 14 creates the event 'e2' and initializes it to be the same as 'e1.' Thi up the **Name** and **Process** of 'e1' (which was set by **expressinterest**); the l will be replaced, but the **Process** is used to direct 'e2' directly back to this cess. In lines 15 – 18, the **Name** of 'e2' is changed to '/Pumpkin' and its **TimeStamp** is set to a quarter of a minute in the future. Then 'e2' is inser the event queue to be delivered when its time arrives (line 19).

Lines 20 – 24 similarly create and initialize another event, 'e3,' and leave the stack for the main loop. The first part of the loop (lines 25 – 27) adds a minute to the **TimeStamp** of the event on the top of the stack (which is ' the first time through), and sends it back for distribution a second later.

This leaves two events in the event queue which this process will be intere when they are eventually distributed: the short-term 'Tick' / 'Tock' a which cycles every second, and the '/Pumpkin' event waiting for 15 seco expire. The **awaitevent** on line 28 will block until one or the other is deli If the event that arrives is one of the copies of 'e3,' its **Name** is translated matching process and printed in line 29 and the event is left on the stack f another trip around the loop (and distribution cycle). When the '/Pumpki event finally arrives, its associated procedure executes as **awaitevent** retu minating the loop, and thus the 'clock' process.

4

Extensibility through POSTSCRIPT Language Files

Extensibility through POSTSCRIPT Language Files **31**

4.1. Server Initialization 31
4.2. File Organization 31
POSTSCRIPT language Files Loaded at Initialization 31
`init.ps` 31
`systoklst.ps` 32
`colors.ps` 32
`cursor.ps` 32
`statusdict.ps` 32
`icon.ps` 32
`util.ps` 32
`litemenu.ps` 32
`demomenu.ps` 32
`litewin.ps` 32
`liteUI.ps` 32
POSTSCRIPT language Files for User Settings 32
`user.ps` 32
`startup.ps` 32
Other POSTSCRIPT language Files 33
`compat.ps` 33
`liteitem.ps` 33
`litetext.ps` 33
`debug.ps` 33

`eventlog.ps` 33
`journal.ps` 33
`repeat.ps` 33
4.3. Some POSTSCRIPT language Extensions in the `*.ps` Files 33
Miscellaneous 33
4.4. User Interaction and Event Management 36
4.5. Rectangle Utilities 38
4.6. Graphics Utilities 39
4.7. CID Utilities 40
4.8. Text and Font Utilities 41
4.9. Journalling Utilities 42
Journalling Internal Variables 43
4.10. Constants 43
4.11. Key Mapping Utilities 43
4.12. Repeating Keys 44
4.13. Colors Definitions 45
4.14. Logging Events 45
UnloggedEvents 45

4

Extensibility through POSTSCRIPT Language Files

There is much more to producing a useful, full-featured window systems platform than the extensions to the POSTSCRIPT language described in the previous two chapters; an ideal server would provide some support for tailoring the user interface, selections between processes, the creation of windows and menus, etc.

This is where the extensibility inherent in the POSTSCRIPT language and NeWS has a tremendous impact. The NeWS server includes these extra features, but they are not ''hard-wired'' into the server; instead, they are provided as sets of POSTSCRIPT language procedures in ASCII files that the server loads, usually when it starts up.

You can look at these files, and are encouraged to do so: the files are in `$NEWSHOME/lib/NeWS`, where `$NEWSHOME` is the directory where you have mounted NeWS (usually `/usr/NeWS`). What is more, you can redefine or replace any of the procedures in those files, either globally when the server starts up, or within a single process.

The next several chapters explain some of the facilities implemented in these files; this chapter gives an overview of them, and lists some of the miscellaneous procedures they define.

4.1. Server Initialization

When you start up the NeWS server, its default initialization procedure (see the `news_server(1)` manual page) simply executes the POSTSCRIPT language in an ASCII file called `init.ps` and then runs the POSTSCRIPT language procedure '&main' that `init.ps` defines (and which `user.ps` may have redefined). `init.ps` in turn loads other POSTSCRIPT language files.

4.2. File Organization

Here is an outline of the organization and contents of these POSTSCRIPT language files.

POSTSCRIPT language Files Loaded at Initialization

`init.ps`

Initializes the frame buffer; defines certain primitives in the POSTSCRIPT language rather than C; defines some ''oughta-be'' primitives such as **case**; defines and starts the server; sets certain constants and system defaults; loads most of the initialization files described here. `init.ps` also defines a default ''root'' menu'' from which you can invoke common tools such as a terminal window and a clock. This rootmenu is one of the things you will probably want

to redefine; see Appendix A, *Using NeWS*, for tips on customizing your

`systoklst.ps` — A list of POSTSCRIPT language primitives that are contained in the system c tionary **systemdict**.

`colors.ps` — Implements the X.10V4 lib/rgb colors set. Adds the dictionary **/Colordict** t **systemdict**.

`cursor.ps` — Builds a dictionary useful for naming characters in cursorfont, a special fon cursors. Client-defined cursor fonts can also be built; see Chapter 16, *Font Tools*, for more information.

`statusdict.ps` — Adds a **statusdict** to **systemdict** for users needing extreme printer compatil The file `statusdict.ps` implements the **statusdict** dictionary and its printer-specific operators such as **printername** and **setsccbatch**, as specifie Section D.6 of the *PostScript Language Reference Manual*. Many of these operators are pseudo-implemented, since they have no meaning in a windo tem.

`icon.ps` — Builds a dictionary useful for naming characters in iconfont.

`util.ps` — Procedures shared by packages; anything that is used by more than one pac should be defined in here.

`litemenu.ps` — A menu package.

`demomenu.ps` — Contains the demos that appear in the root menu. Some of these are run b cuting a UNIX program with **forkunix**; the code for other demos is part of POSTSCRIPT language file itself.

`litewin.ps` — A window manager package.

`liteUI.ps` — Defines the user interface for NeWS. Runs several other `*.ps` files, which for selections (cutting and pasting), set up an input focus, and load the app ate translation table for your keyboard.

POSTSCRIPT language Files for User Settings

`init.ps` searches for these in the directory you started the NeWS server f and failing that, looks in your home directory. Sample modifications to th given in Section A.6.1, *Modifying the NeWS Server*.

`user.ps` — Private definitions and re-definitions of system and package POSTSCRIPT language words.

`startup.ps` — If `startup.ps` exists, `init.ps` will execute the POSTSCRIPT langua fragments in it before it loads any of the other packages. This lets you mc characteristics of the server that are used before any of the other POSTSCR language files (including `user.ps`) are loaded; for example, **verbose?** I and the **InitPaintRoot** procedure which draws the background on the

framebuffer while NeWS is loading.

Other POSTSCRIPT language Files

There are other files that define extensions to NeWS that are loaded by individual programs rather than by `init.ps`:

`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 the server.

`liteitem.ps`

A simple item package used by `itemdemo`.

`litetext.ps`

A simple text package that is loaded by `liteitem.ps`; it also supports a blinking caret.

`debug.ps`

POSTSCRIPT language procedures used when debugging.

`eventlog.ps`

A small package for monitoring input-event distribution, described under Section 4.14, *Logging Events* below.

`journal.ps`

A package for recording user actions and replaying them in "player-piano" mode, describe below under Section 4.9, *Journalling Utilities*.

`repeat.ps`

Implements variable-rate repeating on keyboard keys. Described under Section 4.12, *Repeating Keys* below.

4.3. Some POSTSCRIPT language Extensions in the `*.ps` Files

Here are some of the more useful and commonly used POSTSCRIPT language procedures in the `*.ps` files listed above. There are many more than are documented here. These procedures are ultimately defined using POSTSCRIPT language operators described in the *PostScript Language Reference Manual* or in Chapter 12, *NeWS Operator Extensions*, of this manual. If these procedures don't do quite what you want, look at their source and define your own versions.

Miscellaneous

case and **append** are operations that nearly all POSTSCRIPT programs need to perform. **sprintf/printf/fprintf** are near equivalents to their UNIX counterparts. **arrayinsert**, **arraydelete** and **arrayop** are useful operations on arrays. **dictbegin/dictend** save you from counting the size of dictionaries. **modifyproc** brackets a procedure. **createcanvas** is a quick way to create a canvas. **sleep** allows a POSTSCRIPT language procedure to sleep for an arbitrary period, and **getvalue** and **setvalue** are useful for checking the status of an item. **errored** is very helpful for graceful error-handling.

case

value {key proc key key proc...} **case** –
Compares *value* against several keys, performing the associated procedure i match is found. The key **/Default** matches all values. The following conve number to a (whimsical) string:

```
MyNumber {
   1 {(One)}
   2 {(Two)}
   3 4 5 {(Between 3 & 5)}
   /Default {(Infinity)}
} case
```

getvalue

– **getvalue** value
Returns the ItemValue. The value depends on the nature of the object (e.g. a button, it is an boolean).

setvalue

value **setvalue** –
Takes value and sets ItemValue to agree with it. Value is dependent on th nature of the object.

append

obj1 obj2 **append** obj3
Concatenates arrays, strings, and dictionaries. In case of duplicate dictiona keys, the keys in the second dictionary overwrite the first's.

sprintf

formatstring argarray **sprintf** string
A utility similar to the standard C `sprintf`(3S). *formatstring* is a string '%' characters where argument substitution is to occur. Thus:

```
(Here is a string:%, and an integer:%) [(Hello) 10] sprintf
```

puts the string

```
(Here is a string:Hello, and an integer:10)
```

on the stack.

printf

formatstring argarray **printf** –
Printing form of **sprintf**. Prints on standard out, like **print**.
See also: **dbgprintf**

fprintf

file formatstring argarray **fprintf** –
Prints to *file*. For example:

```
console (Server currenttime is:%n) [currenttime] fprintf
```

will print the time the NeWS server has been running on your console.
See also: **console**

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]

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]

arrayop

A B proc **arrayop** C
Performs *proc* on pairs of elements from arrays *A* and *B* in turn, placing the result in array *C*. For example:

[1 2 3] [4 5 6] {add} arrayop ⇒ [5 7 9]

modifyproc

proc head tail **modifyproc** {head proc tail}
Adds a *head* and *tail* modification to a procedure. Mainly used to over-ride the behavior of a procedure. Thus:

```
/myproc myproc {(myproc called\n) print} {} modifyproc store
```

modifies the existing version of 'myproc' to print 'myproc called' each time it is invoked.

NOTE *Any of the procedures can be keywords.*

createcanvas

parentcanvas numx numy **createcanvas** canvas
Creates *canvas*, a child of *parentcanvas*, located at (0, 0) relative to its parent and with the given width and height.

dictbegin

– **dictbegin** –

Combined with **dictend**, creates a dictionary "large enough" for subsequent and puts it on the dictionary stack. Avoids guessing what size dictionary to create.

dictend

– **dictend** dict

Returns the dictionary created by a previous **dictbegin**; together, they "shri wrap" a dictionary around your **defs**. Usage:

```
/MyDict dictbegin
   /myvar 1 def
   ...
dictend def
```

sleep

interval **sleep** –

sleep sends itself an event timestamped *interval* in the future, and returns w that event is delivered. *interval* is in minutes, with 16 bits of fraction. The usable resolution is about 10 milliseconds.

errored

any **errored** bool

errored acts just like the **stopped** primitive, but for errors. Because this is erally what **stopped** has been used for, **errored** is recommended.

Using **errored** also allows the debugger to work properly. Thus, if yo currently using **stopped** as a way to detect errors, simply replace it with **er**

4.4. User Interaction and Event Management

The procedures **getanimated**, **getclick**, **getrect**, and **getwholerect** are used `litewin.ps` (and hence in most windowing applications) to let the user i cate window positions on the screen. You can use **forkeventmgr** and **eventmgrinterest** to handle forking an event manager that deals with partic events. Use **setstandardcursor** to set a canvas's cursor to one of the stand NeWS cursors.

getanimated

x0 y0 procedure **getanimated** process

Forks a process that does animation while tracking the mouse, returning th cess object *process* to the parent process. Each time the mouse moves, the cess executes 'erasepage x0 y0 moveto,' pushes the current mouse coor nates *x* and *y* onto its stack, and calls *procedure*. The variables *x0*, *y0*, *x*, a are available to *procedure*. After *procedure* returns, the process executes t **stroke** operator. Thus, your *procedure* can use *x0*, *y0*, *x*, and *y* to build a p that will be drawn each time the mouse is moved — drawing a line to the c cursor location, for example.

The process calling your *procedure* exits when the user clicks the mouse; i leaves the final mouse coordinates in an array '[x y]' on top of its stack, so they are available to the parent process via the **waitprocess** operator. Sinc **erasepage** is executed each time the mouse is moved, the current canv

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.

For example, the following code fragment animates a rubber band line that starts at (100,100) and returns the chosen endpoint:

```
currentcanvas createoverlay setcanvas  % Set current canvas to an overlay.
100 100 { x y lineto } getanimated     % Fork a process to track the mouse
                                       % and draw a line from 'x0, y0' to it.
waitprocess aload pop                  % Wait for the animation to complete,
                                       % then unpacks the returned 'x, y' onto
                                       % the stack.
```

This slightly more complicated version of the **getanimated** call prompts for a circle with its center at (100,100). The mouse controls its radius:

```
100 100 {
    newpath x0 y0
    x x0 sub dup mul y y0 sub dup mul add sqrt    % Compute radius.
    0 360 arc
} getanimated
```

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**] on the stack.

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 the both the origin and the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [**x0 y0 x1 y1**] on the stack.

forkeventmgr

interests **forkeventmgr** process
Forks a process that expresses interest in *interests*, which may be either an array or a dictionary whose values are interests. Each interest must contain, in its /**ClientData** field, a dictionary having an entry (/proc) which is executed by the event manager process. This procedure is called with the event on the stack.

NOTE *The event manager uses some entries of the operand stack; do not use* **clear** *to clean up the stack in your* 'proc' *procedure.*

eventmgrinterest

eventname eventproc action canvas **eventmgrinterest** inter[illegible]
Makes an interest. Suitable for use by **forkeventmgr** or **expressinterest**. F
example:

```
/MyEventMgr [
    MenuButton {/popup MyMenu send}
    /DownTransition MyCanvas eventmgrinterest
] forkeventmgr def
```

will create an event manager that handles popping up a menu.

setstandardcursor

primary mask canvas **setstandardcursor** -
Sets *canvas*'s cursor to the cursor composed of the *primary* and *mask* keyw
primary and *mask* must be cursors in cursorfont, the font of standard systen
sors loaded by `cursor.ps`. For example:

```
/hourg /hourg_m MyCanvas setstandardcursor
```

sets the cursor in 'MyCanvas' to an hourglass, usually to indicate that its p
cess will not be responding to user input for a while.

Here are the cursors (and their masks) in cursorfont:

Table 4-1 *Standard NeWS Cursors*

Primary Image	*Mask Image*	*Description*	*When/Where*
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
nouse	**nouse_m**	no cursor	

See also: **setcanvascursor**

4.5. Rectangle Utilities

rect, **rectpath**, **rect2points** , **rectsoverlap**, and **insetrect** are useful for ma
rectangular coordinates and paths; other graphics procedures are below un
Graphics Utilities.

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.

rect2points

x y width height **rect2points** x y x' y'
Converts a rectangle specified by its origin and size to a pair of points specifying the origin and top right corner of the rectangle.

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.

rectsoverlap

x y w h x' y' w' h' **rectsoverlap** bool
Returns true if the two rectangles overlap.

insetrect

delta x y w h **insetrect** x' y' w' h'
Creates a new rectangle inset by *delta*.

4.6. Graphics Utilities

The following are procedures often used to create graphics in canvases: **fillcanvas**, **strokecanvas**, **cshow**, **rshow**, **rectframe**, **ovalpath**, **ovalframe**, **rrectpath**, **rrectframe**, and **insetrrect**.

fillcanvas

int/color **fillcanvas** –
Fills the entire current canvas with the gray value or color.

strokecanvas

int/color **strokecanvas** –
Strokes the border of the canvas with a one point edge with the gray value or color. Currently only works for rectangular canvases.

cshow

string **cshow** –
Shows *string* centered on the current location.

rshow

string **rshow** –
Shows *string* right-justified at the current location.

rectframe

thickness x y w h **rectframe** –
Creates a path composed of two rectangles, the first with origin *x,y* and size w
the second inset from this by *thickness*. Calling **eofill** will fill the frame, whi
stroke will create a ''wire frame'' around it.

ovalpath

x y w h **ovalpath** –
Creates an oval path with the given bounding box.

ovalframe

thickness x y w h **ovalframe** –
Similar to **rectframe**, but with an oval.

rrectpath

r x y w h **rrectpath** –
Creates a rectangular path with rounded corners. The radius of the corner ar
r, the bounding box is *x y w h*.

rrectframe

thickness r x y w h **rrectframe** –
Similar to **rectframe**, but with a rounded rectangle.

insetrrect

delta r x y w h **insetrrect** r' x' y' w' h'
Similar to **insetrect**, but with a rounded rectangle.

4.7. CID Utilities

There is a simple CID (Client IDentifier) synchronizer package availab
NeWS utilities. It works by generating a unique identifier that is used to gen
a "channel" for talking to the client and receiving responses from the client;
a synchronized manner.

uniquecid

– **uniquecid** integer
Generates a unique identifier (*integer*) for use with the rest of the package.

cidinterest

id **cidinterest** interest
Creates an interest appropriate for use with **forkeventmgr**. The callback pr
cedure installed in this interest simply executes the code fragment stored in
event's **/ClientData** field. Typical use (in go demo):

DAMAGE_TAG is a client-defined tag, not a "standard" part of NeWS.

```
/repair { % - => - (repair the board)
          /MyCID uniquecid def
          DAMAGE_TAG tagprint MyCID typedprint
          [MyCID cidinterest] forkeventmgr
          waitprocess pop
} def
```

This procedure generates a unique id, then notifies the client to repair the /I
aged board by sending over a DAMAGE_TAG and MyCID. It then fo
cess which listens for code fragments from the client to execute. The

waitprocess waits for one of these fragments to exit the **forkeventmgr** callback loop.

sendcidevent

id proc **sendcidevent** –
Sends a code fragment to a process which was created by the **cidinterest** - **forkeventmgr** usage shown above. For example, the go demo uses the following to respond to the repair procedure above. These calls draw the go board, draw the black & white stones, erase a stone, and exit from the **forkeventmgr** callback loop.

```
cdef draw_board(int id)
   id {draw_board} sendcidevent
cdef black_stone(int id, int x, int y)
   id {outline_color black_color x y stone} sendcidevent
cdef white_stone(int id, int x, int y)
   id {outline_color white_color x y stone} sendcidevent
cdef cross(int id, int x, int y)
   id {x y cross} sendcidevent
cdef repaired(int id)
   id {exit} sendcidevent
```

The *id* used is the one sent along with the DAMAGE_TAG. See Chapter 9, *The C Client Interface* for an explanation of the use of **cdef**.

cidinterest1only

id **cidinterest1only** interest
A special form of **cidinterest** which 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 to respond to mouse buttons which place a single stone using the above drawing fragments.

4.8. Text and Font Utilities

The following utilities help you display and manipulate text: **fontheight**, **fontascent**, **fontdescent**, **stringbbox**, **cvis**, and **cvas**, **findfilefont**.

fontheight

font **fontheight** int
Returns *font*'s height.

fontascent

font **fontascent** int
Returns *font*'s ascent.

fontdescent

font **fontdescent** int
Returns *font*'s descent (as a positive number).

stringbbox

string **stringbbox** x y w h
Returns *string*'s bounding box.

cvis

int **cvis** string
Converts a (small) integer into a one-character string.

cvas

array **cvas** string
Converts an array of (small) integers into a string.

findfilefont

string **findfilefont** font
Reads the font family file named by the string and constructs and returns a fo object that refers to it.

This is the way to have a bitmap font loaded into NeWS after it has started up.

The font will be entered into the FontDirectory under the fontname in the fan file.

4.9. Journalling Utilities

The following utilites allow you to control the journalling mechanism. With mechanism it is possible to to record and play back NeWS user input events. T file `$NEWSHOME/lib/NeWS/journal.ps` implements the following procedures:

journalplay

– **journalplay** –
Begin replaying from the journalling file. The default filename is `/tmp/NeWS.journal`.

journalrecord

– **journalrecord** –
Start 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 journallin file.

Only raw mouse and keyboard events are replayed, so the system should be i the exactly the same state at the beginning of the replay as it was at the start the journalling session — exactly the same windows in the same positions o screen, the same user running the system from the same directory, etc. **journ play** does take care of repositioning the mouse for you.

Journalling Internal Variables

There are a number of internal variables that the journalling utilities use:

- **RecordFile** — the journalling file.
- **PlayBackFile** — same as **RecordFile** initially; this is the file from which playback will take place.
- **PlaySpeed** — multiplier for the base replay time speed.
- **PlayForever** — play forever if true.
- **State** — current state of journalling system.

These variables are explained more fully in the comments of the file `$NEWSHOME/lib/NeWS/journal.ps`. They are defined in the NeWS dictionary, journal.

4.10. Constants

The following are common constants, similar to `#define`'s in C: **console**, **framebuffer**, **nullproc**, **nullstring**and **nulldict**.

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.

nullproc

– **nullproc** procedure

Returns a no-op procedure.

nullstring

– **nullstring** string

Returns an empty string.

nulldict

– **nulldict** dict

Returns an empty, small dictionary.

4.11. Key Mapping Utilities

A key may be bound to a procedure with the **bindkey** procedure. A key may be unbound using the **unbindkey** procedure. The following example binds the string `!make` to key [F8] and assigns the NeWS—SunView selection converters to [F9] and [F10][4]:

[4] The [F10] function key doesn't exist on Sun3 keyboards.

```
/FunctionF8 {
                dup begin
                        /Name /InsertValue def
                        /Action (!make0  def
                end
                redistributeevent
} bindkey

/FunctionF9     (sv2news_put) bindkey
/FunctionF10    (news2sv_put) bindkey
```

and the F9 key may be unmapped with:

```
/FunctionF9 unbindkey
```

The following utilities allow you to bind keys:

bindkey

key arg **bindkey** –
Creates a new process which watches for *key* to be depressed, and executes *
whenever that happens. If *arg* is an executable array, name, or string, it is si
handed to the PostScript interpreter. Otherwise, if it is a string, then

'{ *arg* forkunix }'

is what gets evaluated.

unbindkey

key arg **unbindkey** –
Removes the binding of the arg for the specified *key* (there is no need to cal
unbindkey before rebinding a key to a new value - the new value will repla
old in **bindkey**).

4.12. Repeating Keys

By default the standard typing array (not the function keys or shift keys) re
20 times per second, after a .5 second threshold. The repeating keys behavi
implemented by a standalone repeat-keys package,
`$NEWSHOME/lib/NeWS/repeat.ps`, loaded as part of the Extended I
System started by `init.ps`. The threshold and repeat rate can be adjuste
your preference by modifying two values in the **UserProfile** dictionary; yo
put something like the following in your `user.ps` file to change them:

```
UserProfile begin
  /KeyRepeatThresh  1 60 div 2 div def
  /KeyRepeatTime    1 60 div 12 div def
end
```

4.13. Colors Definitions

ColorDict is a dictionary which contains named colors. It is implemented by `colors.ps`, which is loaded by `init.ps`. The color names are from the lib/rgb values in X.10V4. Here are some examples:

```
/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
```

... where **RGBcolor** simply converts 0 – 255 color values into NeWS colors:

```
/RGBcolor {    % R G B => color
       % (Takes traditional 0 – 255 arguments for R G B)
       3 {255 div 3 1 roll} repeat rgbcolor
} def
```

4.14. Logging Events

The file `eventlog.ps` defines a procedure to turn logging of event distribution on and off, and a dictionary of events which should be excluded from the log. “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 clients using events heavily. It adds **eventlog** and **UnloggedEvents** to **systemdict**. The fields of the event which are printed are **Serial#**, **TimeStamp**, **Location**, **Name**, **Action**, **Canvas**, **Process**, **KeyState**, and **ClientData**. Here’s a sample log message:

```
#300 1.582 [166 161] EnterEvent 1 canvas(512x512,root,parent) null [] null
```

UnloggedEvents

This is a dictionary of event names which are considered uninteresting to an event whose **Name** is found in this dictionary will not be logged. The default definition of **UnloggedEvents** is

```
/UnloggedEvents 20 dict dup begin
   /Damaged dup def
   /CaretTimeOut dup def
%  /EnterEvent dup def
%  /ExitEvent dup def
   /MouseDragged dup def
end def
```

5

The Extended Input System

The Extended Input System **49**

5.1. Building on NeWS Input Facilities 49
5.2. The LiteUI Interface 50
5.3. Keyboard Input 50
Keyboard Input: Simple ASCII Characters 50
Keyboard Input: Function Keys 51
Assigning Function Keys 52
Keyboard Editing and Cursor Control 52
5.4. Selection Overview and Data Structures 53
Selection Data Structures 53
Selections: Library Procedures 54
Selection Events 56
/SetSelectionAt 56
ExtendSelectionTo 57
DeSelect 58
ShelveSelection 58
SelectionRequest 58
5.5. Input Focus 59

5

The Extended Input System

The input mechanisms described thus far provide two things:

- a basic, default user interface, and
- a platform on which to build more sophisticated interfaces, such as SunView's.

The default interface provides a simple ASCII keyboard: characters are delivered when a key goes down; there is no way to be notified when a key goes up, or what the state of non-character shift keys is ([Control], [Shift]). Characters are delivered to the last process to express a global interest in them, or to the canvas under the cursor if there is no global interest. Interfaces that use the mouse are responsible for doing their own mouse tracking and interpretation.

5.1. Building on NeWS Input Facilities

This chapter describes an *Extended Input System* (EIS). It 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 lightweig[illegible] in the server. For example, recognizing events that indicate a change [illegible] 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 mad be done for each client, since some clients will not make selections at all, t apply other interpretations to the same user actions.

5.2. The LiteUI Interface

The *liteUI* implementation provides distribution of keyboard input and ma ment of selections in a style reminiscent of SunView.

Primary, Secondary, and Shelve selections are provided; Put and Get work all of them in the standard fashion. Selections are made when the mouse b goes down, and are always in character units. Keyboard focus may be con either by cursor motion into and out of windows, or by clicking a mouse b to reset the focus. In the latter mode, the *Point* button sets both the focus a Primary selection at the indicated position; the *Adjust* button restores the f a window, at its previous position, and without affecting the Primary selec

There is no multi-clicking to grow a selection, and no dragging a selection the button down. The Find and Delete functions do not yet have any clien so they have not been implemented. These restrictions are simply things r (yet) done in liteUI; the underlying facilities to support them are already i EIS.

Clients of the *liteUI* interface are all lightweight processes running in the N server. Such clients may have two categories of interaction with *liteU*[illegible] keyboard input, and dealing with selections (for example, cutting and [illegible] between windows). In general, a client follows the sequence:

- In an initialization phase, the client declares its interest in various clas activity. These classes include simple and extended keyboard input, a selection processing. In response, the EIS sets up a number of interes (some in the global input handler, some in the client's own process), a records the client in some global structures.
- The client process enters its main loop, which includes an awaitevent of the events it receives will be in response to interests expressed in t tialization calls it made. These events will generally be at a high sem level; translating mouse events into selection actions is done inside E The client will typically have more work to do with these events; for ple, characters may be sent across the communication channel to be p cessed in the client's non-POSTSCRIPT language code. Some of the p ing will require calls back into EIS code; for example, a client will ha inform the system what selection it has made in response to selection
- Finally, when a client no longer requires various EIS facilities, it sho revoke its interests, so that resources do not remain committed when longer needs them.

5.3. 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. This is done by invoking **addkbdinterests** (passing the client canvas as an argument) and then sitting in a loop, doing an **awaitevent** and processing the returned event.

addkbdinterests

canvas **addkbdinterests** [events]
declares the client as 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:

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.

The second interest has **/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 doing selection-pasting to this window, or if function-key strings have also been requested (see below).

The third interest has the array [**/AcceptFocus /RestoreFocus /DeSelect**] in its Name. Events matching this interest inform the client it has gotten or lost the input focus. (**/DeSelect** events referring to the focus will have an **Action** of **/InputFocus**; clients doing selection processing may also receive **/DeSelect** events for other objects besides the input focus, as described below under *Selection Events*.) 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 focus. Clients are always free to ignore them.

A process that is about to exit, or that will continue to exist, but wants no more keyboard input, may revoke an 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. client can choose to receive function-key events, either in the form of a key naming the key that went down, or as a string of the form "ESC[*nnn*z" (th ASCII-standard escape sequence for such keys).

To get the function-keys identified by escape sequences, the client should p client canvas to **addfunctionstringsinterest**.

addfunctionstringsinterest

canvas **addfunctionstringsinterest** event
creates an interest in the function keys, expresses interest in it, and returns t event. As a result, when a function key is depressed, **awaitevent** returns an whose **Name** is **/InsertValue**, and whose **Action** is a string holding the esc sequence defined for that key. Only function-key-down events can be recei by this mechanism. **Addkbdinterests** must also have been called for this p cedure to have any effect.

To get the function-keys identified by name, the client should pass its clien vas to **addfunctionnamesinterest**.

addfunctionnamesinterest

canvas **addfunctionnamesinterest** event
creates an interest in the function keys, expresses interest in it, and returns event. As a result, when a function key is pressed, **awaitevent** returns whose **Name** is a keyword like **/FunctionL7**. By default, both up and dow sitions on the keys are noted; the client may change this by storing **/Down sition** (or **/UpTransition**, if that is what is desired) into the **Action** field of returned interest. **Addkbdinterests** must also have been called for this pro to have any effect.

No special procedure is provided to revoke interests generated by either of two procedures, since passing the interest to the **revokeinterest** primitive suffices.

Assigning Function Keys

You may assign a given procedure to be executed when a specified key go down. See the section on **bindef** in Chapter 4, *Extensibility through POSTS language Files*.

Keyboard Editing and Cursor Control

If the client is passing characters through to a shell or some similar process will do its own translations on them, they should be passed unmodified. B the client is dealing with text directly, it should provide the editing and ca motion facilities defined in the user's global profile. To assist in this, the c may ask for incoming events to be checked for a match against those keyb actions, and converted to uniform editing-events if they do. This is done b ing 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 key combinations for **EditBack***:

EditBackChar **null**
delete the character before the caret

EditBackWord **null**
delete the word before the caret

EditBackField **null**
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 **null**
delete from the caret back to the beginning of the current line

EditBackColumn **null**
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 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.

5.4. Selection Overview and Data Structures

Clients that will make selections and pass information about them to other processes declare this interest via **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 proces[illegible] place in the global input handler, and the rest in client processes. There is a bal repository of data about selections, however, and there are some standar mats for the information stored in that repository and communicated betwe selection clients.

A selection is named by its *rank*; in *liteUI*, the ranks are **/PrimarySelectior** **/SecondarySelection**, and **/ShelveSelection**[5]. For each rank, there is a dict ary containing the information known to the system about that selection. S dictionary will be called a *selection-dict* henceforth. It will have at least th lowing three keys defined:

Table 5-1 *Selection-Dict Keys*

Key	Type	Semantics
/SelectionHolder	process	which process made the selecti
/Canvas	canvas	the canvas in which the selectic was made
/SelectionResponder	null | process	what process will answer reque concerning this selection

If **/SelectionResponder** is defined to **null**, there will be other keys defined dictionary, setting out all available information about that selection. A fev keys have been defined because they are expected to be generally useful. T are listed in the table below. Others may be provided by clients as con[illegible] there is no limit on what consenting clients may say to each other abo[illegible] tion.

Table 5-2 *System-defined Selection Attributes*

Key	Type	Semantics
/ContentsAscii	string	selection contents, encoded as a stri
/ContentsPostScript	string	selection contents, encoded as an executable POSTSCRIPT language ob
/SelectionObjsize	number	n >= 0; for text, 1 indicates a charac
/SelectionStartIndex	number	position of the first object of selectic in its container
/SelectionLastIndex	number	position of the last object of selecti in its container

Finally, communications between clients about selections (that is, request their responses) are formatted as another dictionary, hereafter called a *req dict*. When submitted by the requester, the dictionary will have a key nam each attribute it wants the value of. (It may also contain commands the se holder should execute, such as **/ReplaceContents**.) When received by a s tion holder, a request-dict will contain the keys defined by the requester, p

[5] There is nothing to prevent clients from using other ranks, with names they define themselves. S speaking a rank is simply a key in the **Selections** dictionary.

following two:

Table 5-3 *Request-dict Entries*

Key	Type	Semantics
/Rank	rank	the rank selection 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.

Selections: Library 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:

/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
sends a request (contained in *request-dict*) concerning the *rank* selection. The format of a *request-dict* is described above, in Table 5-3, *Request-dict Entries*. The **/SelectionRequester** and **/Rank** entries will be filled in by **selectionrequest**, which will process the request and return a response. If the indicated selection does not exist, null will be returned. Some keys in the request may not have an answer available. In this case they will be defined to **/UnknownRequest** in the response.

selectionresponse

event **selectionresponse** –
is used by a selection holder to return a response to a selection request. The argument should be a /**SelectionRequest** event that has been processed by t holder. (/**SelectionRequest** events are described below under *Selection Eve* The event will be transformed into a /**SelectionResponse** event and returne the requester.

setselection

selection-dict rank **setselection** –
is used by a process to declare itself the holder of a selection. *Selection-dic* dictionary containing either a definition of /**SelectionResponder**, or of key which provide data about the selection itself, as described above in Table 5 *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 package like *liteUI* than to window clients.

Selection Events

As mentioned above, clients may expect to receive six different kinds of ev concerning the selection. Of these, the **InsertValue** event has already beer described under *Keyboard Input*; its usage in the selection context is exactl same as for function strings. The remaining five events and the appro 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 language 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 a *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 a *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 roc 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 ca vas.

Name:
/ExtendSelectionTo

Action:

dict [	**Rank**	**/PrimarySelection | /SecondarySe**
	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 **/SetSe tionAt** event, and the client response is very much the same. The disti 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 i cated **Rank** to include the indicated position. If **Size** indicates growth, extend both ends as necessary to get them at a boundary of the indicate size. (For example, if **Size** has changed from 1 to 2, a text window mig grow both ends of the selection to ensure that they fall at word boundar Adjust the **PendingDelete** mode or ignore it as the window is editable not.

If there was no selection of the indicated rank, pretend there was an em 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 updat

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 **ShelveSelection** to be the same as the selection whose *rank* is in **Action**, using **setselection** as above.

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 library procedures described above provide an interface for instigating such transfers independent of user actions.

5.5. 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 *Selections: Library 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** will be **null**). Thereafter, the client wil[illegible]e events whose **Names** are ASCII character codes.

This section describes a collection of routines provided to inquire about anc manipulate the focus. These normally will not be called by clients of the w system; rather, they support focus-policy implementations, which then com cate with the clients.

The focus is identified in an array with two elements, a canvas and a proces The canvas will be the *canvas* argument to **addkbdinterests**. The process ' be one which called **addkbdinterests**, and which should be doing an **await** for keyboard events.

setinputfocus

canvas process **setinputfocus** –
The input focus is set to be the canvas – client pair identified by the argume **setinputfocus**.

currentinputfocus

– **currentinputfocus** [canvas, process]
The current input focus is returned by **currentinputfocus**. If there is no cu focus, **null** is returned.

hasfocus

process **hasfocus** bool
Returns **true** or **false** as the indicated process is or is not currently the inpu 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 s the focus and the primary selection in a window. Clicking Adjust will the focus at its last position in this window, without making any selec

/**CursorFocus**
a window will receive the focus when the mouse enters its subtree, an it when the mouse exits. If the mouse crosses window boundaries wh function key is down, a focus change is delayed until all function key: up, and then reflects the current situation.

/**DefaultFocus**
events are distributed as though no EIS were in effect.

6

Classes

Classes **63**

6.1. Packages and Classes 63

6.2. Introduction to Classes 65

6.3. Class 'Foo' 67

References 69

6

Classes

The reader familiar with traditional window systems will have noticed no mention of menus, windows, scrollbars, and the like. This is intentional; NeWS provides the *facilities* to build these higher-level user interface tools without imposing its notion of what these tools must be. Think of NeWS as providing the window management ''kernel'' from which a user interface toolkit may easily be made.

On the other hand, there is a need to provide *some* user-interface tools. Our solution is to provide a small set of user interface packages, written entirely in the POSTSCRIPT language, which you may use as a base for more sophisticated development.

These are implemented using an object-oriented programming[K] scheme that is quite similar to Smalltalk.[J] This scheme has several advantages:

- It is a well-documented standard discussed in several easily obtainable books.
- It is easily and naturally mapped onto the POSTSCRIPT language.
- It formalizes the flexibility and modularity available through use of the POSTSCRIPT language dictionaries.
- There are at least two well-documented class hierarchies for application writing: Smalltalk itself, and MacApp[L] Apple's ''extensible application.''

This chapter discusses NeWS's implementation of Smalltalk's class mechanism. The following chapter, *Window and Menu Packages*, presents two packages, menus and windows, built using this class mechanism. Appendix B, *Class LiteItem*, describes the implementation of a demonstration item class using class-based programming. Some of this information was presented in a tutorial given at the Winter 1986 USENIX Graphics Workshop.[M]

6.1. Packages and Classes

The reader unfamiliar with message-passing, classes, and object-oriented programming might like to browse through the references listed at the end of the chapter. However, many of the essential ideas in class-based systems are similar to the more traditional ''package''- or ''module''-based systems.

Briefly:

- Packages (modules) are replaced by *classes.*
- Procedures in packages are replaced by *methods* in classes.
- Creating package objects is replaced by creating new *instances* of a cla
- Package local and global variables are replaced by *class variables.*
- Object variables are replaced by *instance variables.*

New notions are:

- Classes are ordered into a hierarchy by *subclassing* a new class to a pri one, *inheriting* its methods, instance variables, and class variables.
- Methods are invoked by use of the send primitive. The term *message* used for an invocation of a method with its arguments.
- There is a means of constructing classes that is absent in most language module creation.
- Two new concepts, the *self* and *super* pseudo-variables, are introduced They are used in methods to refer to the object that sent the message ar method's superclass, respectively.

Note: *self* does *not* refer to the method's class, but rather the object that originally caused the method to be invoked. If the method is inherited, self will not be the method's class, for example.

- Unlike POSTSCRIPT language procedures, methods are *compiled* when class is created. Currently this simply resolves self and super, and perf some minor optimizations.

The relationship between an instance and its class and superclass is shown figure below. We have made an instance, 'aFoo,' of class 'Foo,' which is class of class **Object**. An instance has a copy of all instance variables of it superclasses, thus 'aFoo' has those required by both 'Foo' and **Object**. Tl methods known by an instance are stored in the classes in its superclass ch Thus 'aFoo' can only respond to methods residing in 'Foo' and **Object**.

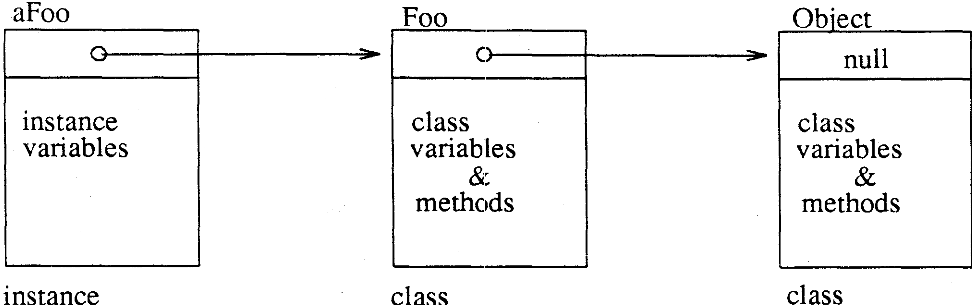


Figure 6-1 *Relationship between Instances and Classes*

Sending a message to an instance requires packaging the arguments to the method, finding the method in the class chain, invoking the method in the context, and possibly returning a result to the sender. If the pseudo-variab is used for the object in sending a message, the search for the method start beginning of the chain, while if **super** is used the search starts in the su

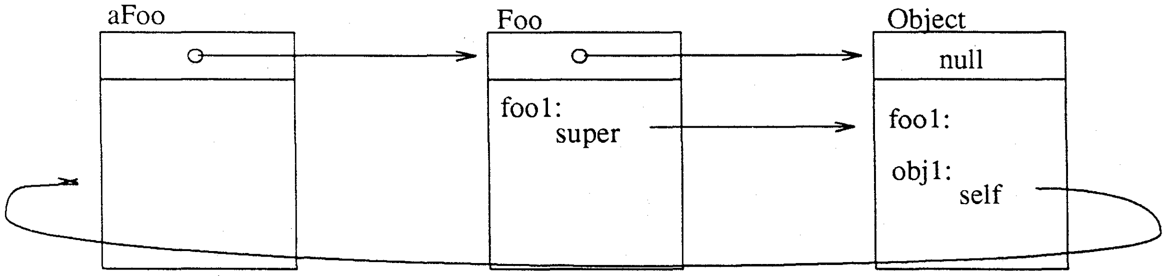


Figure 6-2 *Self and Super*

6.2. Introduction to Classes

The POSTSCRIPT language implementation of classes uses dictionaries to represent the classes and instances. Instances contain all the instance variables of all their superclasses. Classes contain their methods as POSTSCRIPT language procedures. Our current implementation of class is entirely in the POSTSCRIPT language.

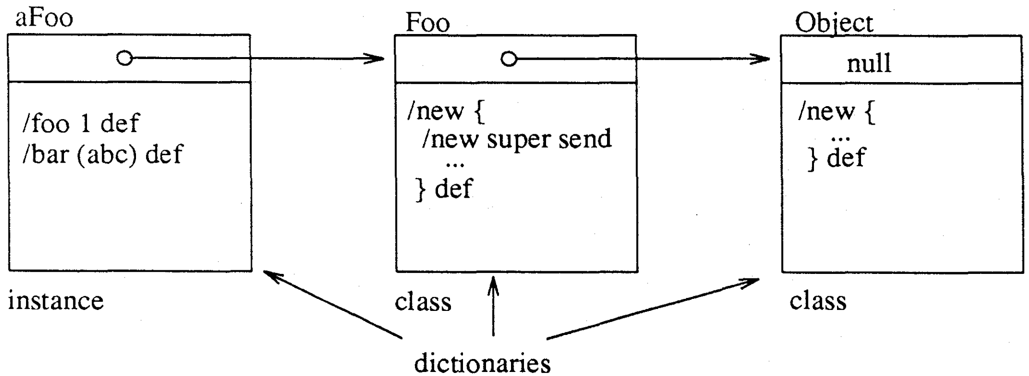


Figure 6-3 *POSTSCRIPT language use of Dictionaries as Objects*

Classes are built using the **classbegin ... classend** procedures; messages are sent with the **send** primitive:

classbegin

classname superclass instancevariables **classbegin** –

Creates an empty class dictionary that is a subclass of *superclass*, and has *instancevariables* associated with each instance of this class. The dictionary is put on the dict stack. *Instancevariables* may be either an array of keywords, in which case they are initialized to null, or a dictionary, in which case they are initialized to the values in the dictionary.

classend

– **classend** classname dict

Pops the current dict off the dict stack (put on by **classbegin** and presumabl filled in by subsequent **defs**), and turns it into a true class dictionary. This involves compiling the methods and building various data structures comm all classes.

send

<optional args> method object **send** <optional results>

Establishes the object's context by putting it and its class hierarchy on the d tionary stack, executes the method, then restores the initial context. The me is typically the keyword of a method in the class of the object, but it can be arbitrary procedure. (See the examples below.)

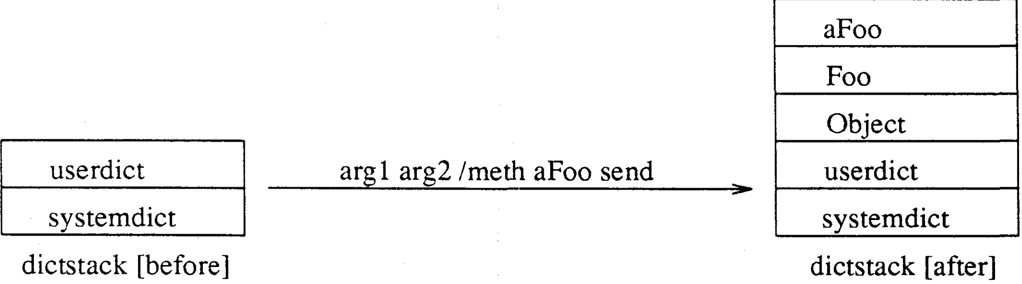


Figure 6-4 *POSTSCRIPT language use of Dictionaries as Objects*

self

– **self** instance

Used as the target object with **send**, **self** refers to the instance that caused th current method to be invoked. It does *not* refer to the class the method is de in. In Figure 6-2, *Self and Super*, the method '`obj1`' in class **Object** uses se refer to '`aFoo`,' not **Object**. The **self** primitive can also be used anywhere refer to the currently active instance.

super

– **super** instance

Used as the target object with **send**, **super** refers to the method being overri by the current method. In the figure above, the method '`foo1`' in class 'Foo overrides the method '`foo1`' in **Object**. The use of **super** in '`foo1`' refers t overridden method. Unlike **self**, **super** cannot be used outside the context o **send**.

Here is the creation of the class **Object**, the root class of all classes:

```
/Object null [] classbegin
% class variables
% methods
    /new {                          % class => instance (make a new object)
        ...
    } def
    /doit {                         % proc ins => - (compile & execute the proc)
        ...
    } def
classend def
```

It is simple indeed, having no instance or class variables, and only two methods at this writing. They are important, however, because they are shared by *all* classes. **/new** builds an instance of a class. If you need to override **/new** to do something unique for your class, you would first call **/new super send** to have your superclasses do their thing; then modify the object you receive. (See the sample below.) **/doit** is used to create a temporary method and execute it in the context of the object. This is generally only required if the procedure contains the pseudo-variables **self** or **super**.

/new may be sent to instances as well as classes. The **/new** method is polymorphic. Exactly the same result is obtained if you send **/new** to a class or an instance of that class.

The **/new** method in class **Object** notices that "self" is an instance rather than a class, and send the message on to its class. This can be a very useful way of creating a new object without knowing its class.

6.3. Class 'Foo'

Now lets build a sample class, 'Foo':

```
/Foo Object                          % 'Foo' is a subclass of Object
dictbegin                            % (initialized) instance variables
   /Value  0 def
   /Time   null def
dictend
classbegin
   /ClassTime currenttime def        % The class variable 'ClassTime'

% class methods
   /new {                            % - => - (Make a new 'Foo')
      /new super send begin
         /resettime self send
         currentdict
      end
   } def
   /printvars {                      % – => - (Print current state)
      (...we got: Value=%, Time=%.\n) [Value Time] printf
   } def
   /changevalue {                    % value => - (Change the value of 'Value
      /Value exch def
   } def
   /resettime {                      % – => - (Change 'Time' to the current t
      /Time currenttime def
   } def
classend def
```

'Foo' is a subclass of **Object**, as discussed above. 'Foo' has two instance v ables unique to each of its objects; 'Value,' an arbitrary value associated wi object, and 'Time,' the time of creation of the object. They are initialized b of the dict form of specifying instance variables. (The **dictbegin ... dicten** are standard utilities that create a dict just the right size for its **defs**.) Foo h one class variable, 'ClassTime,' which is the time of creation of the class.

'Foo' has four methods: 'new,' 'printvars,' 'changevalue,' and 'resettim 'new' first calls its super class to get a raw instance, which it then initializes setting the time to the current time. Note the use of **begin ... currentdict e** This is a s "cliché." Also note the use of both **self** and **super**; we ask our s class to make a new instance of itself and initialize it, then ask self to reset c time. 'printvars' is used to print the instance and class variables of the obje note how this uses another standard utility, **printf**. 'changevalue' is a met that takes a single argument and assigns it to the instance variable 'value.' Finally, 'resettime' sets the instance variable 'time' to the current time.

Let's look at some uses of 'Foo.' Here we create a new instance, 'foo' of 'I We then print out its initial values, shown by the line starting with "...we gc (By the way, we're getting these examples by cutting and pasting into a win running psh onto our NeWS server.)

```
/foo /new Foo send def
/printvars foo send
...we got: Value=0, Time=22.8837.
```

Now we are going to change the value of 'foo's instance variable 'Value.' Note that it initially was an integer, and we are changing it to a string; this is an example of POSTSCRIPT language ''polymorphism.''

```
(A String) /changevalue foo send
/printvars foo send
...we got: Value=A String, Time=22.8837.
```

Similarly, this resets the time value of 'foo.'

```
/resettime foo send
/printvars foo send
...we got: Value=A String, Time=23.1963.
```

Now we do an odd thing, we simply send an executable array (a procedure) to 'foo.' The effect of doing this is to execute the procedure within the context of 'foo.' (This is somewhat unfair, like cutting paper in Origami, but nicely illustrates the flavor of our combination of POSTSCRIPT language features and our class extensions.) The procedure we're sending to 'foo' is **{/Time ClassTime def}**, which assigns 'ClassTime,' the class variable, to 'Time,' the instance variable.

```
{/Time ClassTime def} foo send
/printvars foo send
...we got: Value=A String, Time=22.7444.
```

The above sample did not go through method compilation, thus **self** and **super** could not be used. Let's send an executable to 'foo' to change 'Value' to the number of minutes since its creation, but this time using the more orthodox **doit** method (it uses method compilation).

```
{currenttime Time sub round /changevalue self send} /doit foo send
/printvars foo send
...we got: Value=1, Time=22.7444.
```

Finally, as an extreme example of polymorphism, we set 'Value' to be a procedure returning the current time in seconds, changing over time.

```
{currenttime 60 mul round} /changevalue foo send
/printvars foo send 1000 {pause} repeat /printvars foo send
...we got: Value=1449, Time=22.7444.
...we got: Value=1450, Time=22.7444.
```

References

J. Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, May, 1983.

K. ''Object-Oriented Languages,'' *Byte Magazine*, vol. 11 number 8, McGraw-Hill Inc., August, 1986.

L. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986.

M. Owen Densmore, ''Object-Oriented Programming in NeWS,'' in *Th[illegible]d terey Graphics Workshop*, USENIX, Nov 20-21, 1986.

7

Window and Menu Packages

Window and Menu Packages **73**

7.1. Package Style 73

7.2. A Scrollbar Implementation 74

Description Format 74

7.3. Menu Methods 74

Polymorphic Menu Keys 75

7.4. Window Methods 77

7.5. An example: `lines` 80

7.6. Default User Interface 82

PointButton 82

AdjustButton 83

MenuButton 83

Modifying the User Interface 83

7

Window and Menu Packages

This chapter presents two class-based packages, one for pop-up menus and the other for windows. In addition, we have provided a subclass of the standard window implementation, **ScrollingWindow**. The general notion of a package is discussed, and then the client interface specification for both packages is explained. A detailed sample program follows the explanation. A few hints on modifying the user interface conclude the chapter.

To emphasize that these are "lightweight" implementations, these classes are called **LiteMenu** and **LiteWindow**. Their implementation files are `litemenu.ps` and `litewin.ps` For a more detailed explanation of the **ScrollingWindow** subclass see the section **7.2**, *A Scrollbar Implementation* in `$NEWSHOME/lib/NeWS`.

7.1. Package Style

To create a new instance of a class, use the /**new** method. To create a new instance of class 'Foo' and to assign it to the variable 'foo,' use:

/foo *arg1 ... argN* /new Foo send def

where *arg1 ... argN* are parameters used in initializing the new instance. The needed arguments vary from class to class. The /**new** method is the only method we describe that is sent to a class, rather than to an instance of a class.

Many classes have several subclasses that can be used in place of the class itself. Thus one might have three subclasses of class 'Foo' that can be used anywhere 'Foo' can. For example, suppose we have three interface styles: 'SunViewFoo,' 'MacFoo,' and 'NewsFoo.' To allow the user to specify a preference, we use a variable, 'DefaultFoo,' that has assigned to it the current preference:

/DefaultFoo SunViewFoo store

Note the use of **store** rather than **def**; 'DefaultFoo' already exists and you want to change its value, not possibly create a new one in the current dictionary. The initial values for **DefaultMenu** and **DefaultWindow** are **LitePullRightMenu** and **LiteWindow**, respectively. There are demo root menu entries that let you flip the default window and menu styles between a *SunView*-style and a NeWS-style set. This is done by simply changing the **DefaultMenu** and **DefaultWindow** definitions.

7.2. A Scrollbar Implementation

ScrollingWindow is a subclass of **LiteWindow**. It is defined in `$NEWSHOME/lib/litewin.ps`. There are two simple scrollbars in th frame margin. Two classes, **ScrollbarItem** and **SimpleScrollbar**, provid basis for this implementation. The former is an abstract superclass which the structure of scrollbars, but does not entirely implement one. **SimpleScrollbar** implements a simple, one button scrollbar. The two scrollba initialized to return values between 0 and 1. When viewing a typical docu this corresponds to a position within the document, where 0 indicates the b ning of the document, 1 is the end, and a fraction is somewhere in betwee

Forking A Process

A theme often encountered in the design and use of NeWS packages is fork processes to perform tasks, especially if they may have considerable durat the menu package, for example, a process is forked to track the user intera with the menu, allowing other processing to occur in parallel. Similarly, t dow package will fork a process to redraw the client canvas so that other p ing can occur and so that the drawing itself may easily be interrupted. In t `fish` demo program, for example, you may perform other tasks while the being drawn, or you may interrupt the drawing of the fish to change the si the window or to redraw the fish with different parameters.

Description Format

The interface descriptions below differ somewhat from the usual interface descriptions; only the arguments to the method are given, rather than the e description of the **send**. Thus, if the **send** for method 'samplemethod' lo like:

arg1 ... argN /samplemethod Foo send

the interface description will look like:

/samplemethod

arg1 ... argN **/samplemethod** results

with no mention of the object 'Foo' or the operator **send**.

7.3. Menu Methods

Menus associate a key, generally a string, with an action to be performed w that key is selected by the user. If the menu action is another menu, neste displaying of that menu is performed. The default user interface style use *SunView*-like pull-right nesting.

Notice that the action procedure is more like the *SunView* notifier than a n traditional menu invocation that puts up the menu and waits for a returned This is quite intentional. NeWS favors greater use of multiple lightweight processes. While the menu is being tracked, other computing may also be formed. In particular, NeWS menus do not freeze or lock the screen. This sidered bad manners unless there is good reason for such locking behavior

Polymorphic Menu Keys

Menu keys may be strings, icon names, procedures, or class instances. The string and icon name simply display the corresponding object. In addition, the menu keys may be wrapped in an array. This allows for font and color changes, and slight adjustments in the x,y location of the key relative to its default position. It also allows passing additional arguments to the user's procedure or class instance. Thus

```
[/MyIcon 1 0 0 rgbcolor .5 .5]
```

is a key that shows 'MyIcon' in red with a slight (.5 .5) offset.

The colornames demo has examples of advanced menu key usage.

The following methods are used with menus.

/new

array -or- array array **/new** menu

Creates a new menu. Sent to a menu class, generally **DefaultMenu**, to create an instance of the class. Typically, you use a single argument, which is assumed to be an array of key/action pairs. Thus:

```
/MyMenu [
        (Key 1)     {menuproc1}
        (Key 2)     {menuproc2}
        (Other =>) MyOtherMenu
        ...
] /new DefaultMenu send def
```

creates 'MyMenu,' which displays the strings

```
(Key 1), (Key 2), (Other =>), ...
```

and associates the actions

```
{menuproc1}, {menuproc2}, MyOtherMenu, ...
```

with these keys. If the action is a procedure, it is executed when the user chooses the associated key. If the action is another menu, it is treated as a pull-right menu, and nesting will occur.

If a double argument is used, it is assumed to be an array of keys and an array of associated actions, respectively. If the second array is smaller than the first, it is 'padded' with the last entry. This is mainly used when a single action will suffice for several keys. Thus:

```
/PointSizeMenu
        [ (10) (12) (14) (18) (24) ]
        [ {SetPointSizeFromMenu} ]
        /new DefaultMenu send def
```

might be used as a pull-right menu for setting the point size of a font. The procedure will need to make use of the current menu selection using either **/currentkey** or **/currentindex** below. For example:

```
/SetPointSizeFromMenu { /PointSize currentkey cvi def } def
```

sets the point size to be the integer value of the selected key. Note that we did

not need to use /**currentkey self send** in this situation. The action pr[illegible] called within the scope of the menu instance, and thus does not need to re-establish the menu's context.

/popup

– **/popup** –
Pop-up the menu and track user actions; call the associated action if a key [illegible] selected. This is generally called from an event manager such as:

/**popup** is a direct replacement for the original implementation of /**show**. It does a /**showat** using the current cursor location.

```
/eventmgr [
      MenuButton {/popup MyMenu send}
      DownTransition MyCanvas eventmgrinterest
] forkeventmgr def
```

See Chapter 4, *Extensibility through POSTSCRIPT Language Files*, for desc tions of **eventmgrinterest** and **forkeventmgr**.

/showat

x y *or* event **/showat** –
Pop-up the menu and track user actions. A polymorphic method which can take a pair of integer values for location (*x y*) or an *event* whose location is

/currentkey

– **/currentkey** key
Returns the selected key when called within a menu action procedure.

/currentindex

– **/currentindex** index
Returns the index of the selected key when called within a menu action pr cedure. The indices begin at zero.

/searchkey

key **/searchkey** index bool
Searches for the given key's position in the menu and returns the index of if found (along with a boolean of true) or simply the boolean false if the ke not found.

/searchaction

action **/searchaction** index bool
Searches for the given action's position in the menu, returning a boolean o and its location if found, false otherwise.

/insertitem

index key action **/insertitem** –
Insert a new key/action pair into an existing menu at the given index. Thu

```
0 (Do My Thing) {DoMyThing} /insertitem MyMenu send
```

would add a new menu entry to the top of 'MyMenu.' If *index* exceeds th of the menu, the item is inserted at the end of the menu.

/deleteitem

index **/deleteitem** –
Remove the *index*-th menu item in the menu. Thus:

```
0 /deleteitem MyMenu send
```

would remove the topmost menu entry in 'MyMenu.' If *index* exceeds the size of the menu, the last menu item is deleted.

/changeitem

index key action **/changeitem** –
Replace the menu item at the given index with a new key/action pair. Thus:

```
0 (Do My Thing) {DoMyThing} /changeitem MyMenu send
```

would replace the menu entry at the top of 'MyMenu.' If *index* exceeds the size of the menu, the last menu item is changed.

7.4. Window Methods

A window in NeWS is simply a set of canvases and an event manager. The default window style manages a **FrameCanvas**, a **ClientCanvas**, and an **IconCanvas**. It provides two types of user interface management: menu interaction and direct mouse interaction with the window or icon. See the section on default user interface below for details.

In the following descriptions of window methods, *window/icon* means the window or its icon, depending on whether the window is open or closed. If the window is open, the method refers to the window frame. If the window is closed, the method refers to the window icon.

Also, many of the window methods generally are not used by the client directly, but are accessed primarily through the user interface to the window. These methods are identified below with *(UI)*.

/new

canvas **/new** window
Creates a new window. Sent to a window class, generally **DefaultWindow**, to create an instance of the class. The canvas is the parent canvas for the window and is generally **framebuffer**. After creating a window, a client will want to modify the window by changing its drawing routines, adding a client menu, changing its frame or icon label, etc. The client makes these modifications by changing instance variables in the new window (typically, by sending the window an executable array as a method).

First, the standard creation of a window:

```
/win framebuffer /new DefaultWindow send def
```

Then, the modification of the window:

```
{
    /FrameLabel (Hello World) def
    /PaintClient {MyPaintProc} def
    /IconLabel /hello_world def
    /ClientMenu [
        (First Menu Label)    {MyFirstMenuAction}
        (Next Menu Label)     {MyNextMenuAction}
        ...
    ] /new DefaultMenu send def
} win send
```

Table 1-1 contains the instance variables that are commonly modified, along their initial values; there are other instance variables available, but the interf to them is prone to change.

Table 7-1 **LiteWindow** *Instance Variables*

Instance Variable	*Initial Value*
FrameLabel	nullstring
IconLabel	nullstring
IconImage	null
PaintClient	nullproc
ClientMenu	null

/destroy

– **/destroy** –
Destroy the window and its entire process group. (UI)

/reshape

x y width height **/reshape** –

Note: This does *not* force the shape to be rectangular, just to fit *within* the bounding rectangle.

Reshape the window to have the given bounding box.

/reshapefromuser

– **/reshapefromuser** –
Reshape the window to have a new bounding box. The user is prompted for bounding box, and the results are passed to **/reshape**. **/reshapefromuser** is tially called by the client, but is then handled by the window's user interface (UI)

/map

– **/map** –
Make the window/icon visible. Fork the window's event manager if that has already been done. **/map** is initially called by the client, but is then handled the window's user interface. (UI)

/unmap

– **/unmap** –
Make the window/icon invisible. (UI)

/flipiconic

– **/flipiconic** –
Alternate between opened (window) and closed (iconic) state. (UI)

/move

x y **/move** –
Move the window so that its bottom left corner (its origin) is at the coordinates *x* *y* in its parent canvas. (UI)

/paint

– **/paint** –
Repaint the window or icon. If the window is open, **paint** calls both **/paintframe** and **/paintclient**. The default **/Damaged** handler sets the canvas clip to the damage region and calls **/paint** automatically. (UI)

/paintclient

– **/paintclient** –
Repaint the window's client canvas. The default action is to call the instance procedure variable **PaintClient**. It uses the value of **ForkPaintClient?** to control whether it will fork a process to do the repaint (the default). You should make sure your **PaintClient** uses **pause** where appropriate, so that other processes can run while you are painting.

/paintframe

– **/paintframe** –
Repaint the window's frame. This includes the frame label and controls. The instance variable **FrameLabel** holds the label string. Changing this and invoking paintframe will change the frame's label.

/painticon

– **/painticon** –
Repaint the window's icon. The default action is to set the canvas to **IconCanvas** and call **PaintIcon**. This in turn defaults to using **IconImage** and **IconLabel** to paint the icon. Clients typically either replace the **PaintIcon** procedure with one that does all its own drawing, or set either **IconImage** or **IconLabel** and use the default **PaintIcon** procedure. **IconImage** is initialized to null. Setting it to the name of an icon in the system icon font, such as **/hello_world**, will cause that icon to be drawn. **IconLabel** is initialized to the empty string.

/totop

– **/totop** –
Puts the window/icon above all canvases. (UI)

/tobottom

– **/tobottom** –
Puts the window/icon under all canvases. (UI)

7.5. An example: `lines`

The following is an in-depth look at a sample demo program, `lines`, whic creates a window and associates a menu with its client canvas. The prograr draws lots of lines (in color, if you are using a color machine). The menu cc trols the number of lines drawn. Here's the complete program:

```
#! /usr/NeWS/bin/psh
%
%       lines 6.3 87/02/24
%
% Draw a window with a bunch of lines.
% The icon uses the same drawing procedure as the client.
%
/fillcanvaswithlines {                                  % linesperside => -
        gsave
        1 fillcanvas                                    % Paint the background
        0 setgray                                       % Set the default color black
        clippath pathbbox scale pop pop                 % Set scaling to be 0 to 1
        0 1 3 -1 roll div 1 {                           % 0 delta 1 {..} for
                ColorDisplay? {dup 1 1 sethsbcolor} if  % Set color if ColorDisplay
                0 0 moveto 1 1 index lineto stroke      % Draw line to top
                0 0 moveto 1 lineto stroke              % Draw line to side
                pause                                   % Let others run
        } for
        grestore
} def

/main {
        /linesperside 10 def                            % Start with 10 lines per side
        /setlinesfrommenu {                             % - => - (Set linesperside from menu)
                /linesperside currentkey cvi store      % Set linesperside
                /paintclient win send                   % Ask window to draw me
        } def

        /win framebuffer /new DefaultWindow send def    % Create a window
        {                                               % Install my stuff
                /FrameLabel (Lines) def
                /PaintClient {linesperside fillcanvaswithlines} def
                /PaintIcon {10 fillcanvaswithlines 0 strokecanvas} def
                /ClientMenu
                        [ (2) (4) (8) (10) (20) (100) (250) (500) (1000) ]
                        [ {setlinesfrommenu} ] /new DefaultMenu send def
        } win send
        /reshapefromuser win send                       % Shape it

        /map win send                                   % Map the window
                                                        % (Damage causes PaintClient to be called)
} def

main
```

The program is written as a `psh` script that sends the rest of the file to the NeWS server. The program consists of two procedures, 'fillcanvaswithlines' and 'main,' and a call to 'main.'

The 'fillcanvaswithlines' procedure takes as an argument the number of lines per side to draw. An important point to notice here is the use of **pause** in the **for** loop. This allows the lightweight process mechanism to optimize interactive

behavior. Be sure to use **pause** in any part of your program that will po[illegible] take a long time. In this case, if the user has selected 1000 lines, the dra[illegible] could last several seconds. However, there is a cost in using **pause**; a more efficient version of the `lines` example would pause every 10-20 sets of ve tors.

The procedure 'main' initializes the 'linesperside' parameter, defines a me action procedure, 'setlinesfrommenu', and initializes the program's windo

'setlinesfrommenu' sets the 'linesperside' parameter by converting the m key string into an integer. Note the use of **store**; this is necessary because w changing a predefined value in **userdict** in a context potentially having seve other dictionaries on the dictionary stack. We could have used

```
userdict /linesperside currentkey cvi put
```

instead. Finally, 'setlinesfrommenu' causes the client canvas to be repaint sending **/paintclient** to 'win.' We do this rather than simply calling 'fillcan withlines' directly in order to inherit window manager side effects (mainly t use of forking the client paint procedure).

The window initialization consists of creating a default window, installing o modifications, setting its shape from user input, and finally making it visible The modifications we install set the frame label to the string (''Lines''), set t client repaint procedure to call 'fillcanvaswithlines' with the current value 'linesperside', set the icon to draw itself using 'fillcanvaswithlines,' and fi set the client canvas' menu to be one that resets the 'linesperside' para[illegible]te Note that we map the window as our last step. This is intentional — m[illegible]ir should be deferred until all setup is performed.

7.6. Default User Interface

The (*current*) default window frame and icon surface use all three mouse bu The buttons are identified according to the following definitions in `init.p`

Table 7-2 *Window User Interface Button Usage*

Button Name	*Initial Value*
PointButton	LeftMouseButton
AdjustButton	MiddleMouseButton
MenuButton	RightMouseButton

You may want to redefine these according to taste.

PointButton

PointButton operates as follows:

- In the frame ''go-away'' region (top left): causes the window to close.
- In the frame ''stretch'' region (bottom right): lets the user stretch the wi dow by that corner.
- Elsewhere in the frame: brings the window to the top of the window pil
- Anywhere in the icon: opens the window.

AdjustButton

AdjustButton operates as follows:

- Anywhere in the frame or icon: moves the window or icon.

MenuButton

MenuButton operates as follows:

- Anywhere in the frame or icon: pops up the window or icon menu.

Modifying the User Interface

To change the default settings for the packages, simply put lines like these in your `user.ps` file:

```
% Swap the Adjust and Menu mouse buttons.
/PointButton    LeftMouseButton def
/AdjustButton   RightMouseButton def
/MenuButton     MiddleMouseButton def

% Change the font in menus, and box the selected menu item instead of filling it.
DefaultMenu begin
  /MenuFont /Times-Italic findfont 12 scalefont def
  /StrokeSelection? true def
end

% Change the font for frame headers.
DefaultWindow begin
  /FrameFont /Times-Roman findfont 12 scalefont def
end
```

The following code will allow you to swap the mouse buttons for selections in `psterm` as well:

```
UserProfile begin
    /ViewPoint PointButton   def
    /ViewFocus     PointButton    def
    /ViewAdjust    AdjustButton   def
    /ViewRestore   AdjustButton   def
end
```

8

Debugging

Debugging **87**

8.1. Introduction 87

Contacting the Server 87

Starting a Debugging Session 88

8.2. The Debugging Environment 88

Multi-Process Debugging 88

8.3. Client Commands 88

8.4. User Commands 89

8.5. Miscellaneous Hints 93

Aliases 93

Use Multiple Debugging Connections 93

8

Debugging

A primitive NeWS debugging package is available. It allows you to set breakpoints and print to debugging output windows. It also has a simple facility for automatically causing breaks when errors are encountered. This should be considered simply a tentative poke at the problem. Because the debugger is written in the POSTSCRIPT language, users may modify it for their own purposes.

8.1. Introduction

The NeWS debugger is simply the POSTSCRIPT language itself with a few added commands in the file `debug.ps`. This file is not loaded during the standard initialization process; you need to execute the following code to load the debugging commands:

```
(NeWS/debug.ps) run
```

Normally you would do this by including this line in your `user.ps` file; see *Modifying the NeWS Server* in Appendix A, *Using NeWS*, for more information on the `user.ps` file.

Contacting the Server

The usual style of debugging is to create one or more interactive connections to the NeWS server and start a debugger on each. You can contact the server from any shell using the `psh(1)` command; see *Talking Directly to the Server* in Appendix A, *Using NeWS*, for more information about contacting the server. After the connection is made, a POSTSCRIPT language executive must be invoked by typing the **executive** command. To make this a debugging connection, you then run the command **dbgstart**.

NOTE *Typing* **`dbgstart`** *will make the server try to start an executive for you if one is not running already; However, there are situations where this can fail. If problems occur, start the executive before typing* **`dbgstart`**.

Starting a Debugging Session

A typical initial sequence will look like:

```
paper% psh
executive
Welcome to NeWS Version 1.1
dbgstart
Debugger installed.
```

This assumes `(NeWS/debug.ps) run` has already been executed from `user.ps` file. Had this not been done, you would have had to type **`(NeWS/debug.ps) run`** before typing **`dbgstart`**.

8.2. The Debugging Environment

Debugging commands fall into two categories: commands executed from cl programs (client commands) and commands executed as a debugging user (commands). The user commands are those executed from the `psh` connec to the server, while the client commands are those put in the code being debugged.

dbgstart forks a debugger process that is attached to the `psh` connection a ''listens'' for debugger-related events generated by client commands. (Act *all* client commands simply broadcast debugger events to these debugger da mons!) Any client command that causes printing will print in each debuggi `psh` connection.

Multi-Process Debugging

NeWS is a multi-process environment so there is the problem of debugg several processes at one time. The solution the debugger implements is to l each debugging connection maintain a list of processes that are paused for debugging. This list is printed via the **dbglistbreaks** command below. It is printed whenever a new break occurs. Any of the listed breaks can be *enter* using the **dbgenterbreak** command. This swaps the `psh` debugging conte out and replaces it with the paused process. The context currently consists dict stack and operand stack.

8.3. Client Commands

These are the client commands:

dbgbreak

name **dbgbreak** –
Causes the current client to pause, printing the pending breaks in all debugg connections. *Name* is used as a label in the list to distinguish between breal 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:

```
(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)
```

8.4. 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.

These are the user commands:

dbgstart

– **dbgstart** –
Make the current connection to the server a debugger. Required before any 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** above. They are 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, breakpoi
    4: /undefined called from process(4154624, breakpoi
```

The number preceding the colon is the *breaknumber* used in many of the fo ing commands. A number beyond the end of the listing behaves as the last

dbgbreakenter

name/[dict name] **dbgbreakenter** –
Modify the named procedure to call **dbgbreak** just after starting. If *name* i array, it is assumed to be a dict and the name of a procedure in the dict. Th 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/[dict name] **dbgbreakexit** –
Modify the named procedure to call **dbgbreak** just before exiting.
See also: **dbgbreak**

dbgprintfenter

name/[dict name] formatstring argarray **dbgprintfenter** –
Modify the named procedure to call **dbgprintf** with *formatstring* and *argar* just after starting. Note that *argarray* can be a literal array if you want to de evaluation of the arguments until the **dbgprintf** occurs.
See also: **dbgprintf**

dbgprintfexit

name/[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 a literal 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, breakpoi
```

dbgenter

– **dbgenter** –
Enters the last process listed.

dbgexit

– **dbgexit** –
Return to the debugger connection from whatever process you may have en
This is a no-op if no process is currently entered. The following debugger p
tives will call this routine: **dbgcontinuebreak, dbgkillbreak, dbgenterbre**
dbgstop. Thus, **dbgenterbreak** first calls **dbgexit** to insure preserving stat

dbgcopystack

– **dbgcopystack** –
Copies the current operand stack to the process being debugged. This allov
to **dbgenter** a process, modify that copy of the operand stack, and copy it b
the process.

dbgcallbreak

arg clientproc breaknumber **dbgcallbreak** –
Execute **clientproc** in the broken process with *arg* as data. The **clientproc**
itive will be executed (in the client environment) with the *arg* on the stack,
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 t
word in the executable at the given level, and at the given index within that
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.

8.5. Miscellaneous Hints

Here are some miscellaneous tips for debugging.

Aliases

Because the debugger is POSTSCRIPT language-based, 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.

9

C Client Interface

C Client Interface **97**

9.1. How to Use CPS 97

The .cps File 98

The .h File 100

The .c File 100

Comments 101

9.2. Tags, Tagprint, Typedprint 101

Tags 101

Receiving Tagged Packets from NeWS 102

9.3. A Sample Tags Program 103

9.4. Tokens and Tokenization 105

9.5. The CPS Utilities 106

9

C Client Interface

The C to POSTSCRIPT language (CPS) interface has been designed to facilitate interactions between programs written in the C language on the "client" side and the NeWS server on the "POSTSCRIPT language" side[6]. The interface model is of a client program which constructs a NeWS program and then, after opening a connection to the server, transmits the program. This program may make use of the all the built-in features of NeWS (including procedures already defined in the **userdict** and the **systemdict**).

With this code now resident in the NeWS server, the client side program can make calls to the server side — initiating remote execution. The CPS interface defines:

- the format of the `.cps` file
- the functions which establish and close communication with the NeWS server
- a format for passing information between client and server

9.1. How to Use CPS

There are three component files used in the construction of a NeWS client. These are:

- the `.cps` file - containing the POSTSCRIPT language code to be executed within the server,
- the `.h` file - containing the POSTSCRIPT language code in a form recognizable to the C compiler,
- the `.c` file - the client program, which can use POSTSCRIPT programs.

The CPS program acts to convert the contents of the `.cps` file into a form useable by a C program. The POSTSCRIPT language functions (after conversion) can now be called from the C program and execution will take place within the server (on the POSTSCRIPT language side). An explanation of their use may be found in the following section (*The* `.cps` *File*).

The CPS program needs only one argument (though it has a number of options), the name of the file to translate:

---

[6] NeWS is not limited to communication with programs written in C. Other interfaces may be designed relatively easily.

```
x % cps test.cps
```

The input file in the above example is translated by CPS into a header file (a file)[7]. This header file is then #include'd in the client (C) program before compilation. It will contain not only the definitions that you have made but a number of additional functions that CPS provides (see the section 9.5 on *The* (*Utilities* .

It is best to transmit information to the server once and place procedures and variables to be used more than once in a local dictionary. An error will be ge: erated on the POSTSCRIPT language side if you attempt to reference a procedı or variable (other than implementation-associated ones) that you haven't plac in a dictionary.

There is no need to include the standard i/o header file (**stdio.h**) because CPS this already. However, you will need to add the CPS library to your list of libraries searched by the linker. Further, you will need to add the file psio. the compile line (or as a #include) in your file. This may be done at com time with the following command line form:

```
x % cc -I$NEWSHOME/include test.c /usr/NeWS/lib/libcp
```

where the pathname provided to the compiler is the full pathname of the CPS library (if it is not in your current directory).

The .cps File

The .cps file provides the input for the CPS program. The CPS program buil(header file of the proper form for inclusion in a client program written in the language.

The .cps file contains definitions of the following form:

```
cdef macro_name() procedure
```

where the *macro_name* is the name of the macro as you wish to label it withi your client side program. *procedure* is the POSTSCRIPT language procedure tl you wish to invoke. For example, the ps_moveto() procedure is specifie this way:

```
cdef  ps_moveto(x,y)    x y moveto
```

CPS understands how to construct very efficient C code fragments that packa; and transmit POSTSCRIPT language fragments. The arguments to the C proce are inserted where they are referenced in the POSTSCRIPT language fragment. The invocation:

[7] The input file is passed through cpp(1) before it is read by CPS.

```
ps_moveto(10,20)
```

causes this POSTSCRIPT language fragment to be transmitted:

```
10 20 moveto
```

To reduce communication costs, it is best to keep POSTSCRIPT language fragments that will be used often as short as possible. One good way of doing this is by defining POSTSCRIPT language procedures:

```
cdef initialize()
    /draw-dot { 4 0 360 arc fill } def

cdef  draw_dot(x,y)   x y draw-dot
```

Invoking `initialize()` will transmit the definition of the POSTSCRIPT language function draw-dot a single time. Further invocations of the routine `draw_dot` (with a call to `draw_dot(x,y)`) will require the transmission of fewer bytes.

Argument Types

Just as in normal C procedure declarations, the parameters to CPS macros must be given types. The syntax for specifying a type is different: the type name appears preceding the parameter in the parameter list[8]. For example, the previous definition of `ps_moveto()` is equivalent to:

```
cdef ps_moveto(int x, int y) x y moveto
```

Most of the types correspond directly to C types. The following table lists the CPS argument types:

Table 9-1 *CPS Argument Types*

CPS type	*C type*
`int`	`int`
`float`	`float` or `double`
`string`	`char *`
`cstring`	`char *` with an accompanying count of the number of characters in the string. Such a parameter is actually two parameters: the pointer to the string and the count.
`fixed`	A fixed-point number represented as an integer with 16 bits after the binary point. See the description of integer in Table 13-2, *Implementation Limits* of the *PostScript Language Reference Manual.*

[8] The default type of these arguments is `int` (as in the C language).

Table 9-1 *CPS Argument Types— Continued*

CPS type	*C type*
`token`	A special user-defined token. This is described in the section on user tokens.

The .h File

The header (**.h**) file is created by the CPS utility. It should be included in you client C program using the **#include** feature of the C pre-processer. In additi the routines that have been defined in the `.cps` file, this file includes a num of pre-defined POSTSCRIPT language routines (listed in the final section of tl chapter, *The CPS Utilities*).

The .c File

The client side C program is written in much the same fashion as you would write any C program. The functions that have been declared on the server si (with cdefs) are accessible to you on the client side. While the CPS interface definition does the low-level work of passing these values in a form that you program can understand, you will still have to explicitly open and close com munciation with the NeWS server.

Communication with the Server

Communication with the server is handled by three CPS functions. These fu tions manage the low-level work of determining which server to connect to, establishing the connection, requesting execution of POSTSCRIPT language c and severing the connection to the server cleanly. These functions are part o body of functions that define the CPS interface and which are made avail the inclusion of the CPS library during compilation.

Opening a Connection

A connection to the NeWS server is establishing by calling the CPS function `ps_open_PostScript()`. This function returns a `PSFILE` pointer if a nection to the NeWS server is successfully established, otherwise a `0`. The f tion determines which server to connect to by examining the environment v able NEWSSERVER[9].

`ps_open_PostScript()` must be called before any other procedure tha needs to communicate with the server.

Connection Files

Two `PSFILE` pointers, `PostScript` and `PostScriptInput`, are the duits through which information flows between NeWS server and client progr When the client writes to the NeWS server, it is writing on the file represente the pointer, `PostScript`. When it (the client) reads output from the serve is reading the file `PostScriptInput`. All operations on these `PSFILE` pointers are done using the `psio` package, not the standard I/O. See the *psi* manual page for an explanation. Examples of this may be found in Chapter *Complete Example:* `roundclock`.

[9] For a detailed explanation see Chapter 15, *Supporting NeWS from Other Languages*.

Buffered Output to the Server

Output to the NeWS server is buffered in order to provide a more efficient interface mechanism. The contents of the buffer will be sent to the server when the CPS function `ps_flush_PostScript()` is called.

Closing a Connection

The connection to the server is terminated when the CPS function `ps_close_PostScript()` is called. This function should be called before the client program exits. When invoked it causes all NeWS processes running within the server on the client's behalf to be terminated.

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.

9.2. Tags, Tagprint, Typedprint

There are many ways to implement a communication protocol between client and server. In the CPS interface we have chosen to use a tagged packet method. The server side communicates with the client program by packaging information into packets. These packets are tagged with an identifying number and the information passed in them is typed.

The section *The* `.cps` *File* explains how to call a POSTSCRIPT language function from the client side. This tagged packet method may be said to be the complementary half of that system.

Tags

The CPS interface procedures for receiving input from the NeWS server are somewhat more complicated than the procedures presented in the preceding sections that send POSTSCRIPT language fragments to NeWS. The body of a specification has three parts:

1. A label (*name*) with *args* that the client side program can use.
2. An identifier (*tag*).
3. The POSTSCRIPT language routine or code fragment to be associated with the label.
4. A list of variables to receive the values in the reply (*results*).

The syntax of a specification is:

`cdef` *name* (*args*) *POSTSCRIPT language code* => *tag* (*results*)

These three parts correspond to three phases in the client execution of a CPS procedure:

1. Transmission of the POSTSCRIPT language code.
2. Waiting for the return of the tagged reply.
3. Setting any result values from the reply.

The *tag* field is optional (as are the *args* and *results*) fields. Thus, a specification may be brief as well as lengthy. Both of the following specifications are acceptable:

Figure 9-1 *Short Tags Specification*

```
cdef execute()
	makewin
```

which will execute the POSTSCRIPT language routine `makewin` within the server when the `execute()` function is called from the C client, and

Figure 9-2 *Long Tags Specification*

```
#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	% Output the tag.
	typedprint	% Y1 is on the top of the stack,
	typedprint	% then x1. This is why the return
	typedprint	% list is in the opposite order from
	typedprint	% the argument list.
```

The long specification defines a C function called `ps_bbox` that takes as parameters four *pointers* to integers. It sets the integers to the bounding box the current clipping path. When `ps_bbox()` is called it starts by tra it a block of POSTSCRIPT language code to the NeWS server. In this case, the ' path pathbbox' call returns the bounding box of the current clipping regio then transmits back the tag and results.

Receiving Tagged Packets from NeWS

The tag is necessary in the reply to deal with the possibility of multiple asyr nous messages being sent from the server to the client. For example, if the POSTSCRIPT language code that handles menu selections executes this code the user selects something from the menu:

```
MENU_HIT_TAG tagprint
menuindex typedprint
```

then tagged packets will be coming back from the server to the client at time determined by the users interaction, possibly intermixed with replies to requ like `ps_bbox`. The tags let the client side libraries sort out which replies which requests.

To generate a stub for receiving messages like the menu hits in the previous example, you can use `cdef` with a tag and return value list, but without an POSTSCRIPT language code:

```
cdef ps_menu_hit(index) => MENU_HIT_TAG (index)
```

In this case `ps_menu_hit()` will be a function, not a procedure. It will

to see if the first message on the received message queue is a menu hit. If it is, then it will unpack its arguments and return true, removing the tag and arguments from the queue. Otherwise it will return false, leaving the queue alone. If there is nothing in the received message queue, then the function will wait until something is received.

Functions like `ps_menu_hit()` are generally used to construct the basic command interpretation loops of client programs by using a cascade of them in a polling fashion:

```
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 */
        }
}
```

tagprint and **typedprint**

The **tagprint** statement sends the tag 'BBOX_TAG' back to the client. It places the specified value on the input stream (the file `PostScriptInput`) from which the client side retrieves it. The C client has been waiting for such a tag and the subsequent '**typedprint**'s return the coordinates to the client. The '**typedprint**'s can return any variables of the types listed in the above table, *CPS Argument Types*.

9.3. A Sample Tags Program

Following are two short sample programs. Together, they form a pair which will allow you to return a menu choice from the NeWS server side to the C client side. Remember when you create them to label the CPS file as test.cps. The CPS program will create a test.h file that your C program can use.

Figure 9-3 *A Server Side Tags Program*

```
%
% A very simple NeWS client
%

#define SET_GRAYS_TAG      1

cdef initialize()
   /starpath { % x y w h => - (make a star path)
      matrix currentmatrix 5 1 roll          % xfm x y w h
      4 2 roll translate scale             % xfm
      .2 0 moveto .5 1 lineto .8 0 lineto     % xfm
      0 .65 lineto 1 .65 lineto closepath     % xfm
      setmatrix                          % -
   } def
```

```
/FillCanvasWithStar { % stargray fillgray => -
   fillcanvas setshade
   clippath pathbbox starpath fill
} def
/SetStarGrays { % stargray fillgray => -
   SET_GRAYS_TAG tagprint typedprint typedprint
} def


/win framebuffer /new DefaultWindow send def
{
   /FrameLabel (Tag Test) def
   /PaintIcon {.25 .75 FillCanvasWithStar} def
   /PaintClient {1 0 FillCanvasWithStar} def
   /ClientMenu [
      (Black on White)    { 0  1 SetStarGrays}
      (Black on Gray)     { 0 .5 SetStarGrays}
      (Gray on White)     { .5  1 SetStarGrays}
      (Gray on Black)     { .5  0 SetStarGrays}
      (White on Black)    { 1  0 SetStarGrays}
      (White on Gray)     { 1 .5 SetStarGrays}
   ] /new DefaultMenu send def
} win send
/reshapefromuser win send
/map win send

cdef get_grays(float star, float fill) => SET_GRAYS_TAG (fill, star)

cdef set_grays(float star, float fill)
   win /PaintClient {star fill FillCanvasWithStar} put
   /paintclient win send
```

Figure 9-4 *A Client Side Tags Program*

```
/* A very simple NeWS client */

#include "psio.h"
#include "test.h"

main()
{
   float stargray, fillgray;

   if (ps_open_PostScript() == 0 ) {
      fprintf(stderr,"Cannot connect to NeWS server\n");
      exit(1);
   }
   initialize();
   while (!psio_error(PostScriptInput)) {
      if (get_grays(&stargray, &fillgray)) {
         set_grays(stargray, fillgray);
```

```
        } else if (psio_eof(PostScriptInput)) {
            break;
        } else {
            fprintf ("Strange stuff!\n");
            break;
        }
    }
    ps_close_PostScript();
}
```

9.4. Tokens and Tokenization

NeWS provides a facility for establishing and maintaining a token list. The messages that a client program sends to the NeWS server are sequences

CAUTION **Using the features described here is a performance optimization. You are encouraged not to bother with them until you have your application running, and even then only if communication and interpretation overheads are a problem.**

The token list is a very efficient mechanism for the compression of data prior to transmission. The list is variable in length with a maximum dimension of 1056 elements. The first thirty-two (32) elements are tightly compressed, yielding a 1-byte token. The latter 1024 tokens generate two-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[10].

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(`*u*`)`
: Takes the value on the top of the POSTSCRIPT language stack in the server and defines it as the value of the token *u*. In future messages to POSTSCRIPT language, *u* is this value.

`ps_define_value_token(`*u*`)`
: Defines the user token *u* to be the same as the current value of the POSTSCRIPT language variable *u*. In future messages to POSTSCRIPT

[10] It is frequently useful to add font objects to the token list and save the lookup time.

language, *u* is the value that the POSTSCRIPT language variable *u* h[illegible] time `ps_define_value_token()` was called. Future changes to value of the POSTSCRIPT language variable *u*, or its identity as determi by changes in variable scope, have no effect on the definition of the tok

`ps_define_word_token(u)`
: Defines the user token *u* to be the name of the POSTSCRIPT language va *u*. In future messages to POSTSCRIPT language, *u* the POSTSCRIPT lang variable *u*. This binds the token *u* to the name *u*. When it is sent to POSTSCRIPT language, the name *u* is evaluated and its value is used.

The operators that manipulate the token list are listed in the table in the foll section.

9.5. The CPS Utilities

The following utilities are provided for your use when a `.h` file is created l `cps` utility. You may use these functions without defining them on the serv side. This list does not describe the arguments to these functions. You shou look at the header file for the complete form of the function.

Function	*Description*
ps_open_PostScript	open connection to NeWS server
ps_close_PostScript	close connection to NeWS server
ps_flush_PostScript	flush the output buffer
ps_moveto	**moveto**
ps_rmoveto	**rmoveto**
ps_lineto	**lineto**
ps_rlineto	**rlineto**
ps_closepath	**closepath**
ps_arc	**arc**
ps_stroke	**stroke**
ps_fill	**fill**
ps_show	**show**
ps_cshow	**cshow**
ps_findfont	**findfont**
ps_scalefont	**scalefont**
ps_setfont	**setfont**
ps_gsave	**gsave**
ps_grestore	**grestore**
ps_finddef	takes font,scale returns index into token list
ps_scaledef	takes font and returns index into token list
ps_usetfont	takes 'font token' and returns a font object

10

A Complete Example: `roundclock`

A Complete Example: `roundclock` **109**

`roundclock.c` 109

`roundclock.cps` 111

10

A Complete Example: roundclock

Here is a complete example, the clock program. We start with the C program, and then show the CPS definitions it uses.

roundclock.c

The body of roundclock, implemented in the file roundclock.c, consists of a simple timed loop. The timer can be triggered either because the next unit of time has elapsed or because the NeWS server has received a **/Damaged** event. When damage occurs, the UNIX process repaints the entire clock. When the next unit of time has elapsed, the UNIX process unpaints the old hands and paints the new hands.

roundclock has two options. If the -s flag is specified, roundclock displays a second hand. If -f is specified, roundclock paints itself in a fancy way.

```
/*
 * NeWS clock program
 */

#include <stdio.h>
#include <sys/time.h>
#include <sys/ioctl.h>
#include "psio.h"
#include "roundclock.h"
#include <signal.h>

int     fancy_clock = 0;

main(argc, argv)
        char    **argv;
{
        int     show_seconds = 0;
        int     damaged;
        int     second_hand_length = 38;
        int     minute_hand_length = 35;
        int     hour_hand_length = 20;

        int     lmin = -1, lhour = -1, lsec = -1;
        while (--argc > 0) {
                if ((++argv)[0][0] == '-')
```

```
                switch (argv[0][1]) {
                case 'f':
                     fancy_clock = 1;
                     break;
                case 's':
                     show_seconds = 1;
                     break;
                default:
                     fprintf(stderr,"roundclock: illegal option:%s\n", argv[(
                     exit(-1);
                }
           else {
                fprintf(stderr,"roundclock: illegal option:%s\n",argv[0]);
                exit(-1);
           }
     }
     if (ps_open_PostScript() == 0) {
           fprintf(stderr, "No NeWS server\n");
           exit(-1);
     }
     if (fancy_clock) {
           second_hand_length = 10;
           minute_hand_length = 35;
           hour_hand_length = 20;
           ps_fancy_initializeclock();
     } else {
           ps_initializeclock();
     }
     ps_createclock();          /* initialize round clock window */
     while (1) {
           register struct tm *tm;
           long     now = time(0);
           if (damaged) {        /* Redraw the clock face if necessary */
                 ps_redrawclock();
                 damaged = 0;
           }
           tm = localtime(&now);
           ps_white();       /* Clear out the old hands */
           tm->tm_hour = tm->tm_hour * 5 + tm->tm_min / 12;
           if (lmin >= 0) {
                 if (tm->tm_min != lmin)
                       hand(lmin, minute_hand_length);
                 if (show_seconds && tm->tm_sec != lsec)
                       hand(lsec, second_hand_length);
                 if (tm->tm_hour != lhour)
                       hand(lhour, hour_hand_length);
           }
           ps_black();       /* draw the new hands */
           if (show_seconds)
                 hand(tm->tm_sec, second_hand_length);
           hand(tm->tm_min, minute_hand_length);
           hand(tm->tm_hour, hour_hand_length);
           lsec = tm->tm_sec;
```

```
                lmin = tm->tm_min;
                lhour = tm->tm_hour;
                ps_flush_PostScript();
                {       /* Wait for either the next clock tick or a
                         * window damage repair request */
                        int     msk = 1 << psio_fileno(PostScriptInput);
                        int     n;
                        static struct timeval t;
                        now = show_seconds ? 1 : 60 - tm->tm_sec;
                        t.tv_sec = now > 60 ? 60 : now;
                        if (select(32, &msk, 0, 0, &t) > 0) {
                            char    buf[1000];
                            n = read(psio_fileno(PostScriptInput), buf, sizeof buf);
                            if (n > 0)    /* The only input the clock
                                           * ever gets is a damage
                                           * repair request */
                                damaged++;
                            else if (n == 0)
                                exit(0);
                            else
                                perror("read");
                        }
                }
        }
}

hand(angle, radius)
        int     angle, radius;
{
        if (fancy_clock)
                ps_fancy_hand(-angle * 6, radius);
        else
                ps_hand(-angle * 6, radius);
}
```

roundclock.cps

All painting is done with POSTSCRIPT language routines defined with the help of the CPS library. The CPS definitions are contained in roundclock.cps. This file contains definitions of routines to create a round clock, to draw the clock frame and hands in a simple style, and to draw the clock frame and hands in a fancy style.

```
% CPS PostScript definitions to support clocks.

cdef ps_createclock()
    /window framebuffer /new DefaultWindow send def
    {
        /IconLabel (Clock) def
        /FixFrame { (F) print } def
        /PaintClient { (P) print } def
        /ShapeFrameCanvas {
            gsave ParentCanvas setcanvas
```

```
            FrameX FrameY translate
            FrameWidth FrameHeight scale
            .5 .5 .5 0 360 arc FrameCanvas setcanvasshape
            grestore
    } def
        /ShapeClientCanvas { } def
        /CreateClientCanvas {
            /ClientCanvas FrameCanvas newcanvas def
        } def
        /PaintFrame { } def
        /PaintFocus {
            gsave FrameCanvas setcanvas
            KeyFocus? {KeyFocusColor} {FrameFillColor} ifelse setcolor
            calctransform 0 0 40 0 360 arc stroke
            grestore
        } def
    } window send
    /reshapefromuser window send % Shape it.
    /map window send % Map the window.

    window /FrameCanvas get setcanvas
    /calctransform {
            initmatrix initclip
            clippath pathbbox 100 div exch 100 div exch scale pop pop
            50 50 translate
    } def
    /RDC {
            window /FrameCanvas get setcanvas
            damagepath clipcanvas
            calctransform drawclockframe clipcanvas
    } def

cdef ps_white() W
cdef ps_black() B
cdef ps_redrawclock() RDC

%
% ps_hand draws a plain clock hand.
%
cdef ps_hand(rot,rad)
    gsave
       rot rotate 0 0 moveto 0 rad rlineto
       stroke
    grestore

%
% ps_fancy_hand draws a fancy clock hand.
%
cdef ps_fancy_hand(rot,rad)
    gsave
       rot rotate newpath -5 0 moveto 0 0 5 180 360 arc
       0 rad rlineto -5 5 rlineto -5 -5 rlineto closepath
       fill
```

```
    grestore

cdef ps_initializeclock()
    /drawclockframe {
       bordercolor setcolor clippath fill
       0 0 45 0 360 arc
       backgroundcolor setcolor fill
       textcolor setcolor
       12 {
           0 40 moveto
           0 5 rlineto
           stroke
           30 rotate
       } repeat
    } def
    /B {
        textcolor setcolor
    } def
    /W {
        backgroundcolor setcolor
    } def


cdef ps_fancy_initializeclock()
    /drawclockframe {
        .75 monochromecanvas {setgray} {.7 .7 setrgbcolor} ifelse fill
        clippath fill
        0 0 40 0 360 arc
        1 monochromecanvas {setgray} {1 1 setrgbcolor} ifelse fill
        1 monochromecanvas {setgray} {1 0 setrgbcolor} ifelse fill
        0 45 5 0 360 arc fill
    } def
    /B {
        .5 monochromecanvas {setgray} {.6 1 setrgbcolor} ifelse fill
    } def
    /W {
        1 monochromecanvas {setgray} {1 1 setrgbcolor} ifelse fill
    } def
```

11

NeWS Type Extensions

NeWS Type Extensions **117**

11.1. New Objects in NeWS 117
11.2. Objects as Dictionaries 118
11.3. Canvases as Dictionaries 119
11.4. Events as Dictionaries 121
11.5. Graphics States as Dictionaries 123
11.6. Processes as Dictionaries 124
11.7. Shapes as Dictionaries 126
11.8. Object Cleanup 126
Server Function 126
Object Management 127
Error Handling 127
Connection Management 127
Process Management 127
Killing An Application 127
Garbage Collection 128
11.9. NeWS Security 128

11

NeWS Type Extensions

NeWS extends the POSTSCRIPT language with a number of new types. Some are opaque and can be accessed only by their specific operators. Others can be opened and accessed like dictionaries. The keys available in these ''magic dictionaries'' are discussed in the following sections. Types that are opened as dictionaries are not, unless specially marked as such, read-only.

11.1. New Objects in NeWS

canvas

canvas

Canvas objects represent Cartesian coordinate spaces, with arbitrarily shaped boundaries. Each display is represented by a canvas; others may be created and arranged in an overlapping list.

color

color

Color objects represent a color. They can be defined using either RGB or HSB coordinates; they can be compared; and they can be used as a source of paint for the rendering primitives.

event

event

Event objects represent (a) messages between NeWS processes and (b) input events from physical devices. Events can be accessed as dictionaries.

graphicsstate

graphicsstate

Graphics state objects preserve entire graphics states, as defined by the POSTSCRIPT language, in a permanent form. Their only use is to be saved and restored.

monitor

monitor

Monitor objects are used for mutual exclusion. A monitor object has [illegible]a piece of state indicating whether it is locked or unlocked. Processes can us monitors to implement mutual exclusion (for example, to prevent conflicts updating shared data structures).

process

process

Process objects represent lightweight processes in the POSTSCRIPT languag interpreter. They can be accessed as dictionaries.

shape

shape

Shape objects represent paths, as defined by the POSTSCRIPT language, in a manent form. Their only use is to be saved and restored; only the current p may be operated on.

11.2. Objects as Dictionaries

The internal state of some of these new types is accessible as if the object v dictionary; fields in the object are accessed just like keys in a dictionary. E ples of typical usage (these particular keys are discussed later) are:

```
MyCanvas /Color get                % determine if 'MyCanvas' is a colored
or
MyEvent /Name /EnterEvent put      % set the Name in 'MyEvent' to EnterE
```

CAUTION **The use of new type objects as dictionaries has defined behavior o[illegible] f existing keys given for each type. You should not define new keys[illegible]l "dictionaries"; the results are undefined and what happens may chan future implementations.**

The following sections describe the keys of interest in those types that are sible as dictionaries.

NOTE *In the header to the description of each key, the type to the left of the key i cates what values may be assigned to the key, and the type to the right indi what values may be retrieved. For read-only keys, the position to the left c tains '–.'*

11.3. Canvases as Dictionaries

The accessible keys of a **canvas** dictionary are:

TopCanvas
BottomCanvas
CanvasAbove
CanvasBelow
TopChild
Parent
Transparent
Mapped
Retained
SaveBehind
Color
EventsConsumed
Interests

They are dealt with in order below.

TopCanvas

– **TopCanvas** canvas
The current canvas' top sibling. The **TopChild** of the parent canvas.

BottomCanvas

– **BottomCanvas** canvas
The current canvas' bottom sibling.

CanvasAbove

– **CanvasAbove** canvas or null
The sibling canvas immediately above this canvas, or null if no such canvas exists.

CanvasBelow

– **CanvasBelow** canvas or null
The sibling canvas immediately below this canvas, or null if no such canvas exists.

TopChild

– **TopChild** canvas or null
The top child of this canvas, or null if no such canvas exists.

Parent

canvas or null **Parent** canvas or null
The parent of this canvas, or null if the canvas has no parent. Null is returned, for example, for canvases that result from **createdevice**. Setting this field manipulates the window hierarchy.

Transparent

boolean **Transparent** boolean

True if the canvas is transparent, false if it is opaque. An opaque canvas vis hides all canvases underneath it; a transparent canvas does not. A transparen canvas never has a retained image; instead it shares its parents retained imag

Mapped

boolean **Mapped** boolean

True if the canvas is mapped, false if it is unmapped. When a canvas is ma it becomes visible on the screen that its parent is on. When a nonretained w dow is mapped, the region that becomes visible is considered to be damaged

Retained

boolean **Retained** boolean

True if the canvas is retained, false if it is not. NeWS keeps an offscreen cop a retained canvas. If it is on a screen and overlapped by some other canvas, hidden parts of the canvas will be saved. A retained canvas generally perfor much better with most window management operations, like moving and po ping canvases. But the retained image does consume storage. For color dis the cost of retaining canvases is often prohibitive.

SaveBehind

boolean **SaveBehind** –

SaveBehind is a hint to the window system that when the *canvas* is made vi on the screen canvases below it won't be too active and the canvas won't be too long. This is a performance hint only; it does not affect the semantics o other operations. It is generally employed with popups to reduce the c f damage repair when they are removed.

Color

– **Color** boolean

True if and only if the current canvas can support more colors than just blac white, or greyscale.

EventsConsumed

keyword **EventsConsumed** keyword

The event consumption behavior of the canvas is determined by its **EventsConsumed** key, where *keyword* is one of:

/AllEvents
: no events will be matched against canvases behind this one

/MatchedEvents
: events that match an interest on this canvas will not be matched against canvases behind this one

/NoEvents
: events will be matched against interests on canvases behind this one, re less of whether they match.

Interests

– **Interests** array

The interest list for the canvas is returned as an *array* of events. The order of events in the array is the priority order of the interests, highest first. (The **globalinterestlist** primitive returns an exactly similar array of interests for the global interest list – the set of interests expressed with a null Canvas. It is defined in Chapter 12, *NeWS Operator Extensions*).

11.4. Events as Dictionaries

The currently accessible keys of an **event** dictionary are:

Action
Canvas
ClientData
Exclusivity
Interest
IsInterest
IsQueued
KeyState
Name
Priority
Process
Serial
TimeStamp
XLocation
YLocation

Action

object **Action** object

An arbitrary POSTSCRIPT language object, often depending on the value of the **Name**. For keystrokes, the value of **Action** is **/DownTransition** or **/UpTransition**; for mouse motion, **Action** is null, etc.

In an interest, the **Action** may be a number, a keyword, or a string, in which case it is matched exactly against the **Action** of candidate events; or the **Action** may be an array or dictionary. See Chapter 3, *Input*, for more information on interest matching.

Canvas

null or canvas **Canvas** null or canvas

When an event is submitted for distribution (by **sendevent** or **redistributeevent**), this key indicates a restriction on its distribution: the event only matches interests expressed with respect to the given canvas. Certain system events (such as /**Damaged** events) have a canvas specified.

In an interest, the **Canvas** specifies that only events that happen to that canvas will be matched. Null in an interest **Canvas** indicates an interest in all events not explicitly directed to a particular canvas.

ClientData

object **ClientData** object
In either an interest or an event submitted for distribution, this field may hol(
additional information relating to the event. The information is carried alon;
without modification by NeWS.

Exclusivity

boolean **Exclusivity** boolean
Exclusivity is meaningful only for interests, although it may be set and read
any event (since any event may be used as an argument to **expressinterest**).
true, it indicates that an event that matches this interest in distribution shoul(
be allowed to match any further interests.

Interest

– **Interest** event
This read-only key is set in a real event as it is distributed; its value is the int
that the event matched in order to be delivered to its recipient.

IsInterest

– **IsInterest** boolean
This read-only key indicates whether an event is currently on some interest l

IsQueued

– **IsQueued** boolean
This read-only key is true when the event has been but in the input queue an
not yet been delivered.

KeyState

– **KeyState** array
When keyboard translation is on, this array is empty. When translation is of
this array indicates all the keys that were down *at the time the event was dist
buted.* (Normally, `liteUI.ps` is loaded by the server at initialization anc
turns off translation to gain access to the unencoded Sun keyboard.) The arr
actually contains the **Name** values from events that had an **Action** of **/Down
Transition**, and that did not have a subsequent event with the same **Name** a:
Action of **/UpTransition**. In generating this array, the test is executed befoı
down-event, and after an up-event, so a down-up pair with no intervening ev
will not be reflected in the **KeyState** array.

This key is meaningless in an interest.

Name

object **Name** object
An arbitrary POSTSCRIPT language object, generally indicating the kind of e
For example, keystrokes will have numeric **Names** corresponding to the AS(
characters (or the keys) that were pressed. Many other events have keyword
values, such as **/Damaged** or **/EnterEvent**.

In an interest, the **Name** may be a number, a keyword, or a string, in which c
it is matched exactly against the **Name** of candidate events; or it may be an a
or dictionary. See Chapter 3, *Input*, for more information on interest matchi

Priority

number **Priority** number

Priority is meaningful only for interests, although it may be set and read in any event (since any event may be used as an argument to **expressinterest**). Real events are matched against the interests expressed on a canvas in priority order, highest priority first; among interests with the same priority, the most recent is considered first. For these purposes, the global interest list (interests expressed with a null **Canvas**) is treated like the foremost canvas interest list. The default priority is 0; fractional and negative values are allowed, and there are very few circumstances where the priority need be changed at all.

Process

null or process **Process** null or process

In the event queue, this key indicates the only process the event will be delivered to (if any — it must still match on all other criteria). In an interest list, it identifies the process that expressed this interest.

Serial

– **Serial** number

This key is a number that reflects the order in which events are taken off the event queue. For an interest, **Serial** equals the serial number of the last delivered event that matched this interest. This is a read-only key.

TimeStamp

The current nominal resolution of a time value is 2^{-16} seconds (about 0.9 ms) and the maximum interval is 65,536 minutes (about 45½ days).

number **TimeStamp** number

This numeric value indicates the time an event occurred. (A time value is simply the number of minutes since the system started; it may contain a fractional component.) Events in the 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 **XLocation** number

System events are labeled with the cursor location at the time they are generated; this value is used to determine which canvas(es) the event can be distributed to. It is available to recipients and is transformed to the current canvas' coordinate system. This key accesses the X-coordinate of the location. It is ignored in interests.

YLocation

number **YLocation** number

This key accesses the Y-coordinate of the event location; see the explanation under **XLocation** above. It is ignored in interests.

11.5. Graphics States as Dictionaries

Graphics state objects are intended to be opaque. Their only use is to save the graphics state of a process for future re-use by that (or another) process. They are, therefore, not accessible as dictionaries.

11.6. Processes as Dictionaries

The keys that may be accessed in a **process** dictionary are:

DictionaryStack
ErrorCode
StandardErrorNames
ErrorDetailLevel
Execee
ExecutionStack
Interests
OperandStack
State

All of these keys are read-only; attempts to change their values in a process
unregistered errors.

DictionaryStack

– **DictionaryStack** array
The current dictionary stack of the process is returned as an array. The earli
dictionary is array element 0.

ErrorCode

– **ErrorCode** keyword
The current errorcode of the process is returned as a keyword. The set of pc
ble results is

/accept
/dictfull
/dictstackoverflow
/dictstackunderflow
/execstackoverflow
/interrupt
/invalidaccess
/invalidexit
/invalidfileaccess
/invalidfont
/invalidrestore
/ioerr
/killprocess
/limitcheck
/nocurrentpoint
/none
/rangecheck
/stackoverflow
/stackunderflow
/syntaxerror
/typecheck
/undefined
/undefinedfilename
/undefinedresult
/unimplemented
/unmatchedmark
/unregistered
/VMerror

StandardErrorNames

– **StandardErrorNames** array

StandardErrorNames is an array of the names of the standard errors. It is used by **errored** and the debugger, and is available for other programs' use.

ErrorDetailLevel

process **ErrorDetailLevel** –

Controls the amount of detail that is included in an error report. Setting **ErrorDetailLevel** to 0 (the default) gives a minimum of error reporting. Setting it to 1 yields a more descriptive message, to 2 dumps the contents of the dictionary, execution and operand stacks. Setting the detail level is done as follows:

```
currentprocess /ErrorDetailLevel 1 put
```

Execee

– **Execee** object

The object currently being evaluated (i.e., the top of the process' execstack) is returned.

ExecutionStack

– **ExecutionStack** array

The full current execution stack of the process is returned as an array, containing pairs of executable arrays and indices. The latest executable array is element 0 of the array, the "program counter" within it is element 1.

Interests

– **Interests** array

The current interest list of the process is returned as an array. The first element of the array is the event which is the most recently expressed interest in this process.

OperandStack

– **OperandStack** array

The full current operand stack of the process is returned as an array. The earliest object on the stack is element 0.

State

– **State** array

The current execution state of the process is returned as a keyword. The set of possible results is:

/breakpoint
/dead
/input_wait
/IO_wait
/mon_wait
/proc_wait
/runnable
/zombie

11.7. Shapes as Dictionaries

Shape objects are opaque. Their only use is to save the current path of a [illegible] for future re-use by that (or another) process. They are, therefore, not access as dictionaries.

11.8. Object Cleanup

The following sections in this heading discuss how to manage object cleanup connections and processes. These discussions are directed towards applicatio developers.

Server Function

When NeWS starts, it runs the code in `init.ps` called **/server**. This launch process that listens for connections requests on a well-known socket[11]. Each request for a new client forks a new process which is its initial process. With that process is a small code fragment that:

- makes this process a member of a new process group
- builds the client's userdict
- initializes the graphics state.

The following code does all this:

```
100 dict begin initmatrix newprocessgroup
```

Finally, the client code is executed by executing:

```
connection file cvx exec
```

This converts the connection file to executable and then executes it. It return when the socket is closed by the client.

Any client which has made entries in **systemdict** that should be cleaned up should do so before killing the client processgroup. This can be done by ove riding the **DestroyClient** method in the client window or by catching the e on the client socket.

This latter is done by starting a recursive file read in the client process. (See description above of how the client process is initialized) This is done by ha the CPS initialization program include the following:

```
cdef initialize()
    /AbortProc {..do cleanup here..} def
    /NestedServer {currentfile cvx exec AbortProc} def

    /win framebuffer /new DefaultWindow send def
        ...
        NestedServer
cdef ...
```

[11] See the `news_server(1)` manual page for a more complete discussion.

The client will have altered the **/NestedServer** loop to add the additional process **/AbortProc**. This code will be executed in the initial client process when the socket is closed[12].

Object Management

There is no way to determine all the objects that the server thinks are being referenced. The reason for this is that there is currently no way to enumerate all the processes known to the server. Because of this, there is no way to get at the dictionaries that are "private" to these, mainly their **userdict**. Thus the best one can do is enumerate all objects visible to **systemdict**.

There are ways, however, for a given client to allow for its objects to be enumerated. The simplest way is to put hooks in **systemdict**. One way would be to have clients put process objects in a dictionary in the systemdict. Using a dictionary (rather than a composite object) would confer the benefit of precluding duplicate entries. Another way is to have each process listen for special events from other processes[13].

Error Handling

Clients can do much of their own error processing by using their own version of **errordict**. (POSTSCRIPT language error handling is discussed in detail in the *PostScript Language Reference Manual.*) There is a set of standard error names in **systemdict**. There is also a simple utility, **errored** that uses these.

A more subtle use of **errordict** is exhibited by the `debug.ps` debugger.

Client side errors are dependent on their environment. C clients have to do most of their own error-handling. The C client can catch signals and can install a cleanup proc for their server connection using the **AbortProc** and **NestedServer** (discussed above under the section titled *Server Function* .

Connection Management

If a C program is terminated without calling `ps_close_PostScript` the client UNIX process will terminate, closing all its file descriptors. This in turn will close the socket being used by the NeWS server for that process. This will behave much like calling `ps_close_PostScript`, in that the client process in the server will terminate. See the discussion on the initialization of the client process, and the **AbortProc** and **NestedServer** programs.

Process Management

The key to process management is to insure that memory reclamation is efficient. Garbage collection should automatically clean up virtual memory when an application is killed.

Killing An Application

An application should kill its process groups and any other process groups it has created upon its own termination. The NeWS server does this when your client socket closes[14]. The `psh` client creates a new process group so that the closing of the client socket does not automatically kill the client. Otherwise, all `psh` clients would always immediately die as soon as their window was created.

[12] Some resources, such as cached fonts, are considered system resources and are not under client control.

[13] Windows are redrawn in the *lite* tool kit using this method.

[14] If you are running as a simple POSTSCRIPT language only program using the psh shell, the **Zap** menu item will also, by default, kill the its process group.

Garbage Collection

Garbage collection is the process of removing objects from virtual mem[illegible] they are no longer referenced. The problem is quite acute in the current gene tion of printers that understand the POSTSCRIPT language and any virtual memory system must cope with it. Killing the client process group will "der ence" the client's userdict, which will recursively garbage collect all the clie local references. When the main process dies, it kills its process group. Only other process groups have been created is there a need to explicitly kill the f processes made by that group.

When a process dies, the various stacks associated with it are garbage collec (their ref count decremented). The dictionary stack, in particular, will have e entry decremented. If a forked process created a dictionary and put it on its c tionary stack, the only reference will be that of the process' dictionary stack. Thus that dictionary will be garbage collected. Then, any objects in that dict ary will then also be dec-reffed, making them candidates for garbage collecti

De-Referencing Composite Objects

In other words, objects will be garbage collected only when all references to them go away, regardless of the process dying. De-referencing a dictionary references all the objects in the dictionary when the dictionary is garbage co lected. When a composite object (arrays and dictionaries) is garbage collecte each of its elements decreases its refcount and is in turn garbage collected if count has gone to zero.

11.9. NeWS Security

There is a dictionary called **RemoteHostRegistry** maintained in the server v keys are the names of hosts which are allowed to connect to the NeWS server. When NeWS starts up, this just contains the name of the local host. Wheneve connection is attempted, the name of the remote host is checked to see if it is this dictionary, and if it isn't then a message is issued to the user and the con tion is closed.

NOTE *This is exactly the same security that the X window system has.*

A variable in the **systemdict** named **NetSecurityWanted** may be set to false disable this security mechanism.

The shell script `newshost(1)` allows you to manage the registry of permitt host names from the command line.

12

NeWS Operator Extensions

NeWS Operator Extensions **131**

12

NeWS Operator Extensions

^C

– **^C** –
Mercilessly abort the NeWS server. (The name consists of two characters — a '^' followed by a 'C' — not a control-C.)

acceptconnection

listenfile **acceptconnection** file
Listens for a connection from another UNIX process to the NeWS server on *listenfile*. When another process connects, *file* will connect the server with the client. Things that the client writes to the server will appear on *file*, and things written to *file* will be sent to the server. *Listenfile* is created by invoking *file* with the special file name `(%socketl`*n*`)`. *N* is the IP port number that will be used for listening.

activate

proc **activate** process
Creates a new process that will be executing *proc* in an environment that is a copy of the original process's environment. When *proc* exits, the process will terminate. *Process* is a handle by which the newly created process can be manipulated.

See also: **killprocess, killprocessgroup , waitprocess**

arccos

num **arccos** num
Computes the arc cosine in degrees of *num.*

arcsin

num **arcsin** num
Computes the arc sine in degrees of *num.*

arctan

num **arctan** num
Computes the arc tangent in degrees of *num.* You should probably use **atan** instead.

awaitevent

– **awaitevent** event

Blocks the NeWS lightweight process until an event in which it has expre[illegible]d interest happens, and returns an object of type *event* describing it.

See also: **blockinputqueue, createevent , expressinterest , redistributeevent , sendevent**

blockinputqueue

num **blockinputqueue** –

Inhibit distribution of input events from the event queue, until a correspondi **unblockinputqueue** is executed, or *num* minutes, whichever happens first. When calls to **blockinputqueue** are nested, the timeout goes until the last of of them and the queue remains blocked until an **unblockinputqueue** has bee executed to match each **blockinputqueue**.

Events used as arguments to **sendevent** are inserted in the event queue, and hence are subject to inhibition; events passed to **redistributeevent** are not re queued, and so are not inhibited.

See also: **sendevent, unblockinputqueue**

breakpoint

– **breakpoint** –

Suspends the current process.

buildimage

width height bits/sample matrix proc **buildimage** canvas

Constructs a canvas object from the *width, height, bits/sample*, and *proc* par ters in the same way as **image** interprets its parameters. A notable diff[illegible]c between the Sun and Adobe implementations is that if *bits/sample* is 24 ther image is interpreted as color. The inverse of the matrix is used to define the default coordinate system of the canvas.

The parameters represent a sampled image that is a rectangular array of *widt height* sample values, each of which consists of *bits/sample* bits of data (1,8 The data is received as a sequence of characters (i.e., 8-bit integers in the ra to 255). If *bits/sample* is less than 8, the sample bits are packed left to right within a character (from the right-order bit to the low-order bit). Each row i padded out to a character boundary.

The **buildimage** operator executes *proc* repeatedly to obtain the actual imag data. *Proc* must return (on the operand stack) a string containing any numb additional characters of sample data.

The *matrix* parameter specifies the transformation from a unit square to the coordinates of the image.

NOTE *In the current implementation, only matrices of the form [width 0 0 -height height] will work correctly.*

canvastobottom

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

canvastotop

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

clipcanvas

– **clipcanvas** –
The **clipcanvas** operator is similar to **clip** except that it sets a clipping boundary that is an attribute of the current *canvas*, not the current graphics state. This clipping boundary 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.

It is intended to be used when it is necessary to impose clipping restrictions on all operations aimed at a canvas, no matter where they come from. This is typically used during damage repair to restrict updates to the damaged region.

The clip path set by this operation is not the clip path manipulated by the operations **clip, clippath, eoclip**, and **initclip**. The **initclip** operator sets its clip path to the shape of the canvas.

See also: **damagepath, clipcanvaspath**

clipcanvaspath

– **clipcanvaspath** –
Sets the current path to the canvas clipping 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 than the current color. The test takes into account the characteristics of the current device. The standard boolean operators, like `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. For example, you could use this primitive to scroll a text window. The nonzero winding number rule is used to define the inside and outside of the path.

countinputqueue

– **countinputqueue** num

Returns the number of events currently available from the POSTSCRIPT langu
process' input queue. If this number is positive, **awaitevent** will not block.

createdevice

string **createdevice** canvas

Creates a new canvas from a frame buffer device named by *string*. The usua
value for *string* on a Sun is "/dev/fb."

createevent

– **createevent** event

Synthesizes an object of type *event*, which will have null or zero values for a
fields.

See also: **awaitevent, redistributeevent , expressinterest , sendevent**

createmonitor

– **createmonitor** monitor

Creates a new monitor object.

See also: **monitorlocked, monitor**

createoverlay

canvas **createoverlay** canvas

Given a *canvas*, **createoverlay** creates a new *canvas* that overlays the origin
An overlay is like a sheet of cellophane that lays over a canvas. Anything th
drawn in an overlay will float over the underlying canvas. Objects drawn in
overlay will not affect the underlying image, and objects drawn in the under
image will not affect the overlay. Because of the way that overlays are impl
mented on some displays, there will be performance problems if too many th
are written into the overlay. They are intended to be used for animated obje
like rubber band lines and bounding boxes.

The current color is usually ignored when drawing in overlays. They will ge
erally be done in black. This weakness in the specification of overlays is an
explicit feature: it's there to allow overlays to be implemented using a variet
tricks on different types of hardware.

NOTE *In the current implementation, if there are multiple overlays active on the sc
only one of them will be visible, chosen essentially at random.*

currentautobind

– **currentautobind** boolean

Returns true or false depending on whether or not autobinding is enabled for
current process.

See also: **setautobind**

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**, or **sethsbcolor**.

currentcursorlocation

– **currentcursorlocation** x y

currentcursorlocation returns the cursor's position at the time of the last event distributed from the input queue.

currentlinequality

– **currentlinequality** n

Returns an integer between 0 and 1 that represents the desired line quality.

See also: **setlinequality**

currentpath

– **currentpath** shape

Returns an object of type *shape* that describes the current path. It may later be passed to **setpath**.

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 also: **setrasteropcode**

NOTE *The RasterOp combination function exists only to support emulation of existing window systems. If you find yourself using it, you are probably making a mistake and will have problems running your programs on a wide range of displays. The definition of rasterop is display-specific. Currently the* **image** *and* **copyarea** *primitives do not use the rasteropcode.*

currentstate

– **currentstate** state

Returns a **graphicsstate** object that is a snapshot of the current graphics stat

See also: **setstate**

currenttime

– **currenttime** num

Returns a time value *n.nnn* in minutes since some unspecified starting time. only guarantee that is made about the value returned by **currenttime** is that difference of the results of two successive calls is approximately the numbe minutes that have elapsed in the interval of time between them.

damagepath

– **damagepath** –

Sets the current path to be the damage path from the current canvas. The da path will be cleared. The damage path represents those parts of the canvas t were damaged by some manipulation of the scene on the display, and that c not be repainted from stored bitmaps. Processes can arrange to be notified c damage through the input mechanism. Whenever damage occurs to a canva **Damaged** event will be generated.

See also: **clipcanvas**

dumpsys

– **dumpsys** –

Dumps the contents of the system state to the standard output file. Output is quite voluminous and is interesting only to persons who are debugging the server.

emptypath

– **emptypath** boolean

Tests the current path, returning true if it is empty.

enumeratefontdicts

– **enumeratefontdicts** names

POSTSCRIPT language code should use **FontDirectory** in preference to **enumeratefontdicts**.

Scans through all the font dictionaries that NeWS knows about and pushes th family file name onto the stack (of each font family that it can find).

eoclipcanvas

– **eoclipcanvas** –

This is the same as **clipcanvas** except that it uses the even/odd winding num rule rather than the nonzero 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. For example, you could use this primitive to scroll a text window. The even/odd winding number rule is used to define the inside and outside of the path.

See also: **copyarea**

eocurrentpath

– **eocurrentpath** shape

Returns an object of type *shape* that describes the current path using the even/odd rule.

See also: **currentpath**

eoreshapecanvas

canvas **eoreshapecanvas** –

The **eoreshapecanvas** operator is identical to **reshapecanvas** except that it uses the even/odd winding number rule to interprete the path.

See also: **reshapecanvas**

eowritecanvas

file *or* string **eowritecanvas** –

Either opens *string* as a file for writing, or if the argument is a *file* simply writes to it. Creates a rasterfile which 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. **eowritecanvas** would be used to save an image in a file. **eowritecanvas** follows an even-odd winding rule rather than a non-zero winding rule.

See also: **writecanvas, writescreen , eowritescreen**

eowritescreen

file *or* string **eowritescreen** –

Either opens *string* as a file for writing, or if the argument is a *file* simply writes to it. Creates a rasterfile which contains an image of the entire screen. **eowritescreen** writes pixels from the screen, and it will include pixels from canvases that overlap the current canvas. If the current path is empty, the whole canvas is written. **eowritescreen** would be used to do a conventional screen dump. **eowritescreen** follows an even-odd winding rule rather than a non-zero winding rule.

See also: **writecanvas, writescreen , eowritecanvas**

expressinterest

event **expressinterest** –

Input events matching *event* will be queued for reception by **awaitevent**. See Chapter 3, *Input*, for more information on interest matching.

See also: **awaitevent, createevent , redistributeevent , revokeinterest , sendevent**

extenddamage

– **extenddamage** –

Add the current path to the damage shape for the current canvas. A **/Damag** event will be sent to those processes which have expressed interest. Uses th non-zero winding rule.

eoextenddamage

– **eoextenddamage** –

Add the current path to the damage shape for the current canvas. A **/Damag** event will be sent to those processes which have expressed interest. Uses th even-odd winding rule.

file

string1 string2 **file** file

Identical to the Adobe POSTSCRIPT interpreter implementation, with one ex tion: if the file identified by *string1* cannot be found, and it is not an absolut pathname, the server will attempt to open the file `$NEWSHOME/lib/`*strin*

forkunix

string **forkunix** –

Forks a UNIX process to execute *string* as a shell command line. Standard i and output are directed to `/dev/null`.

getcanvascursor

canvas **getcanvascursor** font char char

Gets the cursor identifiers for *canvas*. *Font* is the font where the cursor ima characters (primary and mask) are stored. The first *char* is used as the index locate the primary image and the second *char* is used to locate the mask ma

See also: **setcanvascursor**

getcanvaslocation

canvas **getcanvaslocation** x y

Returns the location of *canvas* relative to the current canvas. *X,y* is a delta (offset) in the current coordinate system from the lower left-hand corner of current canvas to the lower left-hand corner of *canvas*.

See also: **movecanvas**

getenv

string1 **getenv** string2

Returns the value of the variable *string1* in the environment of the server pr as modified by any **putenv** operations. This operator fails with an undefine error if *string1* is not present in the environment. One can guard against thi by using the **stopped** operator to recover from the error. For example:

```
{ (ENV) getenv } stopped { pop (env default) } if
```

geteventlogger

– **geteventlogger** process

Returns the process which is the current event logger, or null if there is none.

getkeyboardtranslation

– **getkeyboardtranslation** bool

getkeyboardtranslation returns a boolean. `true` means the kernel is interpreting the keyboard; `false` means keyboard interpretation is being left to POSTSCRIPT language code, as in **liteUI.**

See also: **keyboardtype, setkeyboardtranslation**

NOTE *Specific to the Sun Operating System Interface; should eventually move into an environment dictionary.*

getmousetranslation

– **getmousetranslation** boolean

Returns true or false as the underlying operating system is or is not doing translation and scaling on the input received from the mouse. Events from the mouse will have the following keyword values in their name fields depending on the value of mouse translation:

Table 12-1 *Mouse Event Translation*

true	***false***
MouseDragged	RawMouseDragged
LeftMouseButton	RawLeftMouseButton
MiddleMouseButton	RawMiddleMouseButton
RightMouseButton	RawRightMouseButton

At present, there is no use for untranslated mouse events.

NOTE *Specific to the Sun Operating System Interface; should eventually move into an environment dictionary.*

getsocketlocaladdress

file **getsocketlocaladdress** string

Returns a string that describes the local address of the *file*. *File* must be a socket file, and will generally be a socket that is being listened to. This is generally used by servers to generate a name that can be passed to client programs to tell them how to contact the server. The format of the string is unspecified.

getsocketpeername

file **getsocketpeername** string

Returns the name of the host that *file* is connected to. *File* must be an IPC connection to another process. Such files are created with either **acceptconnection** or *(%socket) file*. This is generally used with *currentfile* to determine where a client program is contacting the server from.

globalinterestlist

– **globalinterestlist** array

Returns an *array* of events which are the interests currently expressed wını a Canvas field by all processes. The array is in priority order; the first element the array has the highest priority.

hsbcolor

h s b **hsbcolor** color

Takes three numbers between 0 and 1 representing the hue, saturation, and brightness components of a color and returns a *color* object that represents tł color.

imagecanvas

canvas **imagecanvas** –

Renders a **canvas** onto the current canvas. It is much like the **image** operatc except that the image comes from a canvas instead of a POSTSCRIPT languag procedure.

The canvas is imaged into the unit square in user coordinates with (0,0) at th lower left-hand corner and (1,1) at the upper right-hand corner. To image a ‹ vas at a particular place, merely set the CTM to position the unit square, just you would with the **image** primitive.

The **imagecanvas** primitive deals with all scaling and technology mapping issues. It will, for example, map 24-bit color images onto black and white screens by dithering.

imagemaskcanvas

boolean canvas **imagemaskcanvas** –

Renders a **canvas** onto the current canvas. It is much like the **imagemask** o tor except that the image comes from a canvas instead of a POSTSCRIPT lang procedure. The *boolean* determines whether the polarity of the mask canvas be inverted.

The canvas is imaged into the unit square in user coordinates with (0,0) at th lower left-hand corner and (1,1) at the upper right-hand corner. To image a vas at a particular place, merely set the CTM to position the unit square, just you would with the **image** primitive.

insertcanvasabove

canvas x y **insertcanvasabove** –

Inserts the current canvas above *canvas*, using the same interpretation of *[x* **movecanvas**. The current canvas must either be a sibling or child of *canvas* The *mapped* attribute of the canvas does not change.

insertcanvasbelow

canvas x y **insertcanvasbelow** –

Inserts the current canvas below *canvas*, using the same interpretation of *[x,y]* as **movecanvas**. The current canvas must either be a sibling or child of *canvas*. The *mapped* attribute of the canvas does not change.

keyboardtype

– **keyboardtype** number

Returns a small integer indicating the kind of keyboard attached to the NeWS server. The return value is actually the return from the `KIOCTYPE` ioctl, documented under `kb`(4S).

See also: **getkeyboardtranslation, setkeyboardtranslation**

NOTE *Specific to the Sun Operating System Interface; should eventually move into an environment dictionary.*

killprocess

process **killprocess** –

Kills *process*.

killprocessgroup

process **killprocessgroup** –

Kills *process* and all other processes in the same process group.

See also: **newprocessgroup**

lasteventtime

– **lasteventtime** num

Returns the **TimeStamp** of the last event delivered by the input system.

localhostname

– **localhostname** string

Returns the network hostname of the host on which the server is running.

max

a b **max** c

Compares *a* and *b* and leaves the maximum 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 minimum 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 most one process may have a monitor locked at any one time. If a process attempts to lock a locked monitor it will block until the monitor is unlocked. If an error occurs during the execution of *procedure* and the execution stack is unwound beyond the *monitor*, then the *monitor* object will be 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** –

Moves the current canvas to (*x*,*y*) relative to its parent. (*x*,*y*) is a delta vecto interpreted according to the current transformation. This motion is relative lower left-hand corner of the two canvases – (0,0) interpreted with referenc the initial matrix for each canvas. The *mapped* attribute of the canvas does change.

See also: **getcanvaslocation**

newcanvas

pcanvas **newcanvas** ncanvas

Creates a new empty canvas, *ncanvas*, whose parent is *pcanvas*.

These defaults are the result of historical precedent. To ensure the portability of your programs to future releases of the system, you should always explicitly set the **Transparent** property of all new canvases.

It defaults to being opaque if its parent is the framebuffer; transparent other It defaults to being retained if it is opaque and the number of bits per pixel (framebuffer is less that the retain threshold. If your program relies on havir canvas be retained you should explicitly set it to be retained.

See also: **reshapecanvas**

newprocessgroup

– **newprocessgroup** –

Creates a new process group with the current process as its only member. V a process forks the child will be in the same process group as its parent.

pathforallvec

array **pathforallvec** –

The single argument to **pathforallvec** is an array of procedures. The **pathforallvec** operator then enumerates the current path in order, executing one procedures out of the array for each of the elements in the path. The type o path element determines which array element will be executed. **moveto, li** **curveto**, and **closepath**, respectively, are array elements 0, 1, 2, and 3. If tl array is too short, **pathforallvec** will try to reduce elements of one type to another. Array element 5 is used to handle conic control points. The stand POSTSCRIPT language operator **pathforall** is exactly equivalent to '4 array astore pathforallvec.' For further information, consult the *PostScript Lan Reference Manual* description of the **pathforall** operator. Users are caution against using this primitive if at all possible, and using **pathforall** 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.

putenv

string1 string2 **putenv** –

Defines the shell environment variable *string1* to have the value *string2*. The environment variables inherited by the server as modified by **putenv** calls are inherited by UNIX processes created as children of the server with **forkunix**.

random

– **random** num

Returns a random number in the range [0,1].

readcanvas

string *or* file **readcanvas** canvas

Reads a *sun* raster file into a newly created *canvas* (see the Pixrect reference manual). The argument to **readcanvas** should be a *file* object or a *string*. The canvas is either read from the file object or from the file named by the string. The *string* must be the name of a file in the server's file name space. The canvas that is created will be retained and opaque. The canvas will have the depth specified in the raster file, will not have a parent, and will not be mapped. The canvas *cannot* be mapped; an **invalidaccess** error will result if you try to map the canvas. The canvas is useful only as a source for **imagecanvas.** If the *file* can't be found, an **undefinedfilename** error is generated. If the *file* can't be interpreted as a raster file, an **invalidaccess** error is generated.

recallevent

event **recallevent** –

The event passed as an argument is removed from the event queue. The most common use for this primitive is to turn off a timer-event that has been sent but not yet delivered.

See also: **sendevent**

redistributeevent

event **redistributeevent** –

Return an event that has been received by the calling process to the distribution mechanism, which will continue as though the event had not matched the interest which gave it to this process.

See also: **expressinterest**

reshapecanvas

canvas **reshapecanvas** –

Sets the shape of *canvas* to be the current path, and it sets the canvas' defaul transformation matrix from the current transformation matrix. This also est lishes its position relative to the current canvas. If *canvas* is the same as the current canvas, then an implicit **initmatrix** will be done. The entire content the canvas is considered to be damaged.

The **initclip** operation will set the path to the shape defined by the shape of current canvas.

Think of the current transformation matrix as laying down a grid over the cu path. This grid has its origin somewhere relative to the path and it has some scale, rotation, and skew associated with it. When **reshapecanvas** sets the default transformation matrix for the canvas, it sets it so that this same grid laid over the canvas as is laid over the current path, with the origin in the sa relative location.

revokeinterest

event **revokeinterest** –

No more input events matching *event* will be distributed to this NeWS proces

See also: **expressinterest**

rgbcolor

r g b **rgbcolor** color

Takes three numbers between 0 and 1 representing the red, green, and blue ponents of a color and returns a *color* object that represents that color.

sendevent

event **sendevent** –

Submit an event to the input distribution mechanism (i.e., sort it into the ev queue according to its **TimeStamp**). See Chapter 3, *Input*, for more inform about event distribution.

See also: **awaitevent**, **createevent** , **recallevent** , **redistributeevent** , **expressintere**

setautobind

boolean **setautobind** –

Enables or disables autobinding for the current process. By default it is on. Section 13.4, *Autobind*, for more information on autobinding.

See also: **currentautobind**

setcanvas

canvas **setcanvas** –

Sets the current canvas to be *canvas*. Implicitly executes **newpath initmat**

setcanvascursor

font char char **setcanvascursor** –

Sets the cursor identifiers for the currentcanvas. *Font* is the font where the cursor image characters (primary and mask) are stored. The first *char* is used as the index to locate the primary image and the second *char* is used to locate the mask image.

See also: **getcanvascursor**

setcolor

color **setcolor** –

Sets the current color to be *color*. The operation **rgbcolor setcolor** is the same as **setrgbcolor**, and **hsbcolor setcolor** is the same as **sethsbcolor**.

setcursorlocation

x y **setcursorlocation** –

Moves the cursor so its hot spot is at (*x*, *y*) in the current canvas' coordinate space.

seteventlogger

process **seteventlogger** –

Process (which must have expressed some interest – it doesn't matter what) is made to be the event-logger. Thereafter, a copy of every event that enters the distribution mechanism will be given to this process prior to (and without affecting) the rest of the distribution mechanism. This facility is offered as a POSTSCRIPT language debugging aid.

See also: **geteventlogger**

setfileinputtoken

object integer **setfileinputtoken** –

Used to define compressed tokens for communication efficiency. **setfileinputtoken** takes a specified *object* and a specified *integer* and associates them. They are then placed in the token list at the index location specified by the *integer*.

setkeyboardtranslation

bool **setkeyboardtranslation** –

Kernel translation of the keyboard is turned on or off, as the argument is `true` or `false`.

See also: **keyboardtype**

NOTE *Specific to the Sun Operating System Interface; should eventually move into an environment dictionary.*

setlinequality

n **setlinequality** –

Sets the current desired line quality to *n*, which must be a number from 0 to
Line quality controls the quality of lines rendered by the **stroke** primitive.
Increasing values of line quality increase the quality of the rendered line, an
decrease performance. A value of 0 renders lines as fast as possible with the
least attention paid to quality (the line thickness is ignored, lines are always
single pixel wide). A value of 1 renders lines with the highest possible qual
they will be the correct width, and all endcaps and joins will be correct. Inte
mediate values may give you different quality/performance tradeoffs.

The default value for line quality is 1. If the value of line quality is not spec
the lines drawn will be 1/72" wide, independent of your coordinate space.

See also: **currentlinequality**

setmousetranslation

boolean **setmousetranslation** –

Instructs the underlying operating system to switch to the indicated mouse t
lation mode. The initial value is `true`.

NOTE *Specific to the Sun Operating System Interface; should eventually move into environment dictionary.*

setpath

path **setpath** –

Sets the current path from the shape object *path*.

setprintermatch

boolean **setprintermatch** –

Sets the current value of the **printermatch** flag in the graphics state to *bool*
When printer matching is enabled text output to the display will be forced to
match exactly text output to a printer. The metrics used by the printer will b
imposed on the display fonts. This will usually cause displayed text to look
bunched up and generally reduce readability. With printer matching disable
readability will be maximized, but the character metrics for the display will
correspond to the printer.

See also: **currentprintermatch**

setrasteropcode

num **setrasteropcode** –

Sets the current rasterop combination function, which will be used in subse
graphics operations. The values that **setrasteropcode** takes are the same as
RasterOp function codes used by the Pixrect library, though they must be ca
lated: useful values are `PIX_NOT(PIX_DST)` = 5, `PIX_SRC^PIX_DST`
`PIX_SRC|PIX_DST` = 14, etc. See the *Pixrect Reference Manual* for fur
details.

NOTE *The RasterOp combination function exists only to support emulation of exis window systems. If you find yourself using it, you are probably making a m and will have problems running your programs on a wide range of displays definition of rasterop is display-specific. Currently, the* **image** *and* **copyare** *primitives do not use the rasteropcode.*

See also: **currentrasteropcode**

setstate

graphicsstate **setstate** –
Sets the current graphics state from *graphicsstate.*
See also: **currentstate**

startkeyboardandmouse

– **startkeyboardandmouse** –
Initiate server processing of keyboard and mouse input. This is called once from early initialization code in `init.ps`, and should not be called again.

suspendprocess

process **suspendprocess** –
Suspends the given process.
See also: **breakpoint**, **continueprocess**

tagprint

n **tagprint** –
Prints the integer *n* where $-2^{15} \leq n < 2^{15}$
encoded as a tag on the current output stream. Tags are used to identify packets sent from the NeWS server to client programs. See Chapter 9, *C Client Interface*, for information on how the CPS input mechanism uses tags.

typedprint

o **typedprint** –
Print the object *o* in an encoded form on the current output stream. These objects can then be read by client programs using the facilities of CPS. The format in which objects are encoded is described in Chapter 14, *Byte Stream Format.*

unblockinputqueue

– **unblockinputqueue** –
An input queue lock set by **blockinputqueue** is released. If this reduces the count of locks to 0, distribution of events from the input queue is resumed. If the count was already 0, a rangecheck error is raised.
See also: **blockinputqueue**

undef

dictionary key **undef** –
Removes the definition (if any) of *key* from the *dictionary.*

waitprocess

process **waitprocess** value
Waits until *process* completes, and returns the value that was on the top of its stack at the time that it exited.
See also: **fork**

writecanvas

file *or* string **writecanvas** –

Either opens *string* as a file for writing, or if the argument is a *file* simply wr to it. Creates a rasterfile which contains an image of the region outlined by tl current path in the current canvas. If the current path is empty, the whole car is written. **writecanvas** would be used to save an image in a file.

See also: **writescreen, eowritescreen , eowritecanvas**

writescreen

file *or* string **writescreen** –

Either opens *string* as a file for writing, or if the argument is a *file* simply wr to it. Creates a rasterfile which contains an image of the entire screen. **writescreen** writes pixels from the screen, and it will include pixels from canva that overlap the current canvas. If the current path is empty, the whole canva written. **writescreen** could be used to do a conventional screen dump as fol

```
framebuffer setcanvas (/tmp/snap) writescreen
```

See also: **writecanvas, eowritecanvas , eowritescreen**

13

Omissions and Implementation Limits

Omissions and Implementation Limits **151**

13.1. Operator Omissions 151
13.2. Imaging Omissions 151
13.3. Implementation Limits 152
13.4. Autobind 153

13

Omissions and Implementation Limits

13.1. Operator Omissions

The following POSTSCRIPT language primitives were defined by Adobe, but have not yet been implemented in the NeWS POSTSCRIPT language interpreter.

Table 13-1 *Omitted POSTSCRIPT language primitives*

Primitive	*Note*
banddevice	Printer specific.
charpath	Pseudo-implemented.
copypage	Pseudo-implemented.
currentscreen	Pseudo-implemented.
currenttransfer	Pseudo-implemented.
echo	Printer specific.
executeonly	
framedevice	Printer specific.
invertmatrix	
noaccess	
nulldevice	Printer specific.
prompt	Printer specific.
renderbands	Printer specific.
resetfile	
restore	Pseudo-implemented.
reversepath	
save	Pseudo-implemented.
setscreen	Pseudo-implemented.
settransfer	Pseudo-implemented.
showpage	Pseudo-implemented.
start	Replaced by `user.ps/startup.ps` initialization.
translate	Missing matrix argument version.
usertime	

13.2. Imaging Omissions

Two portions of the stencil/paint imaging model remain to be implemented: halftone screens and transfer functions. NeWS pseudo-implements many operators specific to the POSTSCRIPT language printer interface with the **statusdict** dictionary. The release file `statusdict.ps` contains these implementations.

13.3. Implementation Limits

Table 13-2 *Implementation Limits*

Quantity	*Limit*	*Explanation*
integer	32767	Integers are represented as 32 bits, 16 bits of the fraction. Integers are automatically converted to if they overflow.
real		Single-precision floating-point numbers are used Reals are represented as fractional integers if the small enough, but the type determination operatc 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 comn cation channels). The limit is `getdtablesize()`−*n*, where *n* depends on th ticular server but will be about four.
userdict	100	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	100	Maximum function/compound statement nesting
gsave level		Expanded as required.
path		Expanded as required.
VM		The server expands to use as much VM as the ur ing system permits.
interpreter level		Not applicable.
save level		Not applicable.

13.4. Autobind

When the POSTSCRIPT language interpreter encounters an executable name, the interpreter searches the dictionary stack from the top to the bottom until it finds a definition for this name. The execution time will therefore increase as the size of the dictionary stack increases. On the other hand, this method allows one to redefine the behavior of a name by defining it in a dictionary and placing this dictionary on the dictionary stack.

The POSTSCRIPT language provides an operator called **bind** that will circumvent this name lookup process. **bind** goes through a procedure and checks each executable name inside it. If a name resolves to an operator object in the context of the current dictionary stack, then **bind** alters the procedure by replacing the name with the operator object. This eliminates the time taken by name lookups when executing this procedure, but it removes the flexibility of being able to change a procedure's behavior by redefining names before executing it.

NeWS implements an *autobind* mechanism which will cause every executable procedure to behave similarly to the way it would as if **bind** had been called on it.

The following example illustrates the differences among the cases with no binding, using **bind**, and using autobinding.

```
paper% psh
executive
Welcome to NeWS version 1.1
false setautobind
/test1 { 5 3 add == } def
/test2 { 5 3 add == } bind def
true setautobind
/test2.5 { 5 3 add == } def
false setautobind
/add { sub } def
/test3 { 5 3 add == } bind def
true setautobind
/test4 { 5 3 add == } def

test1
2
test2
8
test2.5
8
test3
2
test4
8
```

In this example, the 'test1' procedure calls 'add.' Since 'add' was redefined to be '{ sub },' 'test1' really does a subtraction. The **bind** operator was used on procedure 'test2,' so the 'add' was not redefined as it was for 'test1.' Similarly, 'test2.5' behaves like 'test2' since autobinding was enabled. **bind** was run on 'test3,' but since 'add' now resolves to something other than an operator,

no binding takes place. The 'test4' procedure was defined with autob[illegible]u on, but like 'test3', 'add' resolves to something other than an operator, so ı binding takes place.

NOTE *Autobinding is on by default.*

Autobinding can be turned on or off using the **setautobind** operator, which a boolean argument. You can use the **currentautobind** operator to get the current setting; it returns a boolean value. If you want to redefine the behav a name that is defined in **systemdict**, you should make sure that autobinding off when the name is redefined and when procedures that use the new defini are defined.

14

Byte Stream Format

Byte Stream Format **157**

14.1. Encoding 157
14.2. Object Tables 158
14.3. Magic Numbers 159
14.4. Examples 159

14

Byte Stream Format

The information in this section is only of interest to those 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 which may be freely intermixed with the ASCII encoding. The two encodings are differentiated based on the top bit of the eight-bit bytes in the stream. If the top bit is zero, then the byte is an ASCII character. If it is one, then the byte is a compressed token. This differentiation is not applied within string constants or the parameter bytes of a compressed token.

14.1. Encoding

Each compressed token is a single byte with the top bit set. There may be parameter bytes following it and there may be a parameter encoded in the bottom bits of the code byte. In the following description of the various tokens, the values are referred to symbolically. The mapping between these names and numeric values is given at the end of this chapter.

enc_int

enc_int+(*d*<<2)+*w* ; *w**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

enc_short_string+*w* ; *w**C*

$0 \le w \le 15$: The next w bytes are taken as a string.

enc_string

enc_string+*w* ; *w**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

enc_syscommon+*k*

$0 \le k \le 32$: Inside the 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 — 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

enc_syscommon 2; *k*

0≤*k*≤255: This is essentially identical to **enc_syscommon** except that th index into the object table is *k*+32. This allows the less common POSTSCRIPT language operators to be encoded as two bytes.

enc_usercommon

enc_usercommon +*k*

0≤*k*≤31: This is similar to **enc_syscommon** except that it provides user-definable tokens. Each communication channel to the server has an ass ated POSTSCRIPT language object table. The **enc_usercommon** token causes the *k*th table entry to be inserted in the input stream. The table i dynamic; it is the responsibility of the client program to load objects in this table. The POSTSCRIPT language operator **setfileinputtoken** associ an object with a table slot for an input channel.

enc_lusercommon

enc_lusercommon +*j* ; *k*

0≤*j*≤3 and 0≤*k*≤255: This is essentially identical to **enc_usercommon** e that the index is (*j*<<8)+(*k*+32).

enc_IEEEfloat

enc_IEEEfloat ; 4**F*

The next four bytes, high order to low order, form an IEEE format float point number.

enc_IEEEdouble

enc_IEEEdouble ; 8**F*

The next eight bytes, high order to low order, form an IEEE double pre floating-point number.

14.2. 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 t used as the value of the token. These tokens are typically a part of a POSTS language stream that is to be executed and can be any kind of object. Usual either executable keyword or operator objects are used.

This has some subtle implications with scope rules. If the object is a keywo then its value will be looked up before being executed, just as an ASCII encc keyword would be. If it is an operator object, then the operator will be exec directly, with no name lookup. This improves performance, but it also bind 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, the whenever that token was encountered **moveto** would be looked up and exec On the other hand, if the value of **moveto** were loaded into the slot, then wh ever that token was encountered the interpretation of **moveto** *at the time the was loaded* would be used.

14.3. Magic Numbers

Here is the binding between token names and values:

Table 14-1 *Token Values*

Value	*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	5	free
0260	32	enc_syscommon
0320	32	enc_usercommon
0360	16	free

14.4. Examples

The POSTSCRIPT language fragment:

```
10 300 moveto
(Hello world) show
```

can be encoded simply as an ASCII text string:

```
"10 300 moveto\n(Hello world) show "
```

which would give 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 14-2 *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
0261	**moveto** — assuming that **moveto** is in slot one of the operator table, which it isn't
0233	(0220+11) start of an 11-character string
0110	'H'
0145	'e'
...	
0144	'd'
0262	**show** — assuming that **show** is in slot two of the operator table, which it isn't

15

Supporting NeWS From Other Languages

Supporting NeWS From Other Languages **163**

15.1. Contacting the Server 163
15.2. Communication with the Server 163

15

Supporting NeWS From Other Languages

As it comes out of the box, the only language that is supported for NeWS clients (besides the raw POSTSCRIPT language) is C. The CPS preprocessor is primarily responsible for providing C support. What CPS and the `libcps.a` library provide is a mechanism for contacting the server (`ps_open_PostScript()`) and a mechanism for creating and sending messages to NeWS on that I/O channel.[15]

15.1. Contacting the Server

To contact the server from a UNIX environment you first need to get the environment variable `NEWSSERVER`. This contains a string like `3227656822.2000;paper`. The first number is the 32-bit IP address of the server in host byte order. The second number is its IP port number. You need to create a socket and connect it to this IP address and port. Following the semicolon in `NEWSSERVER` is the text name of the host on which the server is running, which you can ignore.

The `setnewshost(1)` command is a shell script that fabricates the appropriate string for `NEWSSERVER`.

Once a connection has been established, all you need to do is write bytes down the stream as described in the Chapter 14, *Byte Stream Format*. Remember that you don't need to use the compressed binary tokens, they are merely an optimization. It is perfectly satisfactory to send ASCII POSTSCRIPT language code with no compression.

15.2. Communication with the Server

Eventually, a CPS-like program that is appropriate for the language should probably be written. The basis for such a program would be the input and output facilities that CPS uses; the program could write routines that called them, or macros that expanded into invocations of them, or whatever other technique suited the host language.

There is a C function called `pprintf()` which is the runtime output workhorse behind CPS. It is invoked in a manner identical to `fprintf()`(3s), with a format string that is interpreted in the same way. When values are output with `%s` or `%d` or any of the other formatting specifiers, they are output as compressed binary tokens. The rest of the format string is output as is; it may

[15] The `psh`(1) and `say`(1) programs provide these mechanisms to users who want simply to send POSTSCRIPT programs to the server.

contain compressed tokens or simple ASCII.

Input from NeWS to the client appears as bytes that can be read from the serv
I/O stream. The format of these bytes is entirely up to the POSTSCRIPT lang
code downloaded by the client into the server, so it may be as simple or as co
plex as you wish. There is a facility in NeWS for writing objects back to the
clients using the same compressed binary format as the client uses to send to
server and a corresponding C procedure `pscanf()` for interpreting these m
sages.

16

Font Tools

Font Tools **167**

16.1. Cursor Fonts 167
A Standard Font 167
Representation 167
Format 168
Generating a Font 168
16.2. Building an Ordinary Font 170

16

Font Tools

This chapter describes how to create bitmap fonts and place them in the NeWS font library. It is like a cookbook; its level of detail is only good enough to create simple fonts such as cursor fonts and icon fonts, and to convert existing fonts into NeWS's format. It is not a complete description of the font format. The font utility programs described herein are all subject to change. With all these caveats aside, let us proceed.

16.1. Cursor Fonts

This section covers the creation of cursor fonts.

A Standard Font

NeWS supplies the beginnings of a standard cursor font. The files involved are:

```
$NEWSHOME/lib/NeWS/cursor.ps
$NEWSHOME/fonts/Cursor.ffam
$NEWSHOME/fonts/Cursor12.font
```

($NEWSHOME is the location of your NeWS installation, usually /usr/NeWS.) The intent is to build this font into a collection of well-done generic cursor images. As you develop such images you are encouraged to send them to Sun to be included in the standard cursor font. Some screening will be applied to weed out non-generic cursors. The advantages of using the standard cursor font involve a more uniform look between applications and greater resource sharing within the NeWS server.

Representation

Cursors are normally implemented as bitmap fonts. The closest approximation to the requested font will be selected from the font library of bitmaps.

Thus, cursor font shape descriptions are just bitmaps. The primary image is rendered in black over a white mask image. There is no such thing as an XOR cursor.

If a user-defined font is used, the shape descriptions may be arbitrary POSTSCRIPT language. Cursors may have dynamic color, image, transformations, and shape. However, the cursor shape descriptions are called at the server's discretion, not on every mouse motion.

Format

The font utility `mkiconfont` expects input in the format illustrated b[illegible] examples below. This happens to be the format generated by the *SunV[illegible]t* `iconedit` program, but can be generated in any manner. Here is an exam a cursor named *pointer* that is used for the root window in the default `ini` file. Its image is that of a narrow arrow that points up and to the left.

```
/* Format_version=1, Width=16, Height=16, Depth=1, Valid_bits_per_item=16
 * XOrigin=0, YOrigin=15
 */
    0x0000,0x4000,0x6000,0x7000,0x7800,0x7C00,0x7E00,0x7800,
    0x4C00,0x0C00,0x0600,0x0600,0x0300,0x0300,0x0180,0x0000
```

`XOrigin` and `YOrigin` indicate the origin of the character, which is the spot of the cursor. The values for `XOrigin` and `YOrigin` originate in t bitmap's lower left-hand corner with positive values extending up and to th right. `YOrigin` is strange in that it starts from the last non-zero row of pi not the bottom of the bitmap. When using `iconedit`, `XOrigin` and `YOrigin` need to be entered by a separate text editor.

Here is another example of a cursor named *right_arrow* that is used by the r package supplied by the `menu.ps` file. Its image is that of an arrow that p right.

```
/* Format_version=1, Width=16, Height=16, Depth=1, Valid_bits_per_item=16
 * XOrigin=17, YOrigin=6
 */
    0x0000,0x0020,0x0030,0x0038,0x003C,0x7FFE,0x7FFF,0x7FFE,
    0x003C,0x0038,0x0030,0x0020,0x0000,0x0000,0x0000,0x0000
```

Note that it is OK for the origin of the character to be off the edge of the bitr

Cursors have a mask image and a primary image. Here is the mask for the *pointer* cursor. It is called *pointer_mask*.

```
/* Format_version=1, Width=16, Height=16, Depth=1, Valid_bits_per_item=16
 * XOrigin=0, YOrigin=16
 */
    0xC000,0xE000,0xF000,0xF800,0xFC00,0xFE00,0xFF00,0xFF80,
    0xFE00,0xDF00,0x9F00,0x0F80,0x0F80,0x07C0,0x07C0,0x03C0
```

Note that the mask image is used to outline the primary image and thus its c is offset by one from the primary image so as to superimpose the images correctly. This is typical of cursor masks.

Generating a Font

Here is the process for generating a simple font:

1. Generate a collection of ASCII bitmap file pairs (see the *Format* section above). The convention is to call each cursor *name*`.cursor` and its n *name*`_mask.cursor`. Create a file containing these file names, each on a separate line. In this example, the file is called *myfont.list* — you name it whatever you want. The pair order should be primary file nam lowed by mask file name.

2. Make an ASCII version of the font from the list of ASCII bitmap files usi program `mkiconfont`. The first argument to `mkiconfont` i[illegible]l

containing a list of file names. The second argument to `mkiconfont` is the name of the output file prepended by a > and the intended name of the font family.

```
x% mkiconfont myfont.list MyFont>MyFont12.afb
```

3. Turn the ASCII version of the font into a binary version using the program `dumpfont`. The first argument should be a `-d` flag. The second argument is the directory in which the resulting `.fb` file should be placed (no spaces between the flag and argument). The third argument is the name of the file of the ASCII version of the font. The output file is named like the ASCII version but with a `.fb` suffix instead of a `.afb` suffix.

```
x% dumpfont -d. MyFont12.afb
```

4. Build a font family file for the font using the program `bldfamily`. `bldfamily` will look in the current directory to find the font files.

```
x% bldfamily -d.
```

5. To reference the font symbolically, one can build a `.ps` file that contains a dictionary of character names for the font. Here is an example of the way to do this:

```
#! /bin/sh
egrep "^(STARTCHAR|ENCODING)" MyFont12.afb>myfont.ps
ed - myfont.ps<<'EOF'
g/STARTCHAR/j
1,$s'STARTCHAR *)ENCODING *)'/1 /2 def'
1i
/myfontdict 300 dict def
myfontdict begin
$a
end
% Usage: x y moveto /myfontname showmyfont
/showmyfont {
        currentfont ( ) dup 0 myfontdict 5 index get put
        myfontfont setfont show setfont pop } def
/myfont (MyFont) findfont 12 scalefont def
w
q
EOT
```

6. Copy the font and font family files to the font directory.

```
x% cp MyFont.ff MyFont12.fb $NEWSHOME/fonts
```

7. Copy the `.ps` to a well known place.

```
x% cp myfont.ps $NEWSHOME/lib/NeWS
```

8. Use the .ps file at the front of the font in your POSTSCRIPT program

```
(NeWS/myfont.ps) run
myfontdict begin
myfont name name_mask setcanvascursor
end
```

16.2. Building an Ordinary Font

In the POSTSCRIPT language a font, like **Times-Roman**, is a scale-able obj font file containing a set of bitmaps is an instance of a POSTSCRIPT langua at some particular size and orientation. Because of the special requiremen the POSTSCRIPT language, NeWS has its own font file format. A group of tl font files, called a *family* can be used to implement a POSTSCRIPT languag

There are two steps to create a POSTSCRIPT language font from a set of for

1. the font files must first be converted into NeWS format using dumpfo
2. then a description of them as a family must be build using bldfami

dumpfont(1) will take a set of Adobe ASCII format or *vfont*(5) format fil convert them to NeWS format with file extension .fb. bldfamily(1) w a set of NeWS fonts and build a font family file with extension .ff. For a description of these programs and their options, see their manual pages.

For example, say you have a set of *vfont* files gacha.b.7, gacha gacha.b.12, and gacha.b.14; and you would like them to appear i as the POSTSCRIPT font **Gacha-Bold**. Call dumpfont with this comman

```
x% dumpfont -d$NEWSHOME/fonts -S -n Gacha-Bold gach
```

dumpfont will convert the files named gacha.b.* into NeWS format, v them into the $NEWSHOME/fonts directory (usually /usr/NeWS/fc calculate their size information by inspecting the bitmaps, and force the na **Gacha-Bold**. Then call bldfamily with this command line:

```
x% bldfamily -d$NEWSHOME/fonts
```

bldfamily will scan $NEWSHOME/fonts for files named Gacha-Bold*n*.fb. It will then build a font family file and write it to $NEWSHOME/fonts/Gacha-Bold.ff. Now you can send the POSTS language code:

```
(Gacha-Bold) findfont
```

to the NeWS server to pick up the font family that you have built. You can POSTSCRIPT language **scalefont** primitive to select from the different size maps.

Using NeWS

Using NeWS **173**

A.1. NeWS Environment Variables 173

Which Server Binary? 174

The Debugging Server Binary 174

A.2. Starting up NeWS 174

From outside `suntools` 174

From within `suntools` using `overview(1)` 174

Server Initialization 174

A.3. SunView1 Binary Compatibility with NeWS 174

Bugs in SunView1/NeWS Coexistence 175

Inconveniences 175

Screen Damage 176

Input Mismatches 176

NeWS on the Sun-3/110 176

A.4. Learning NeWS 177

Putting A Message in a Window 177

The `psh` Command 177

Running POSTSCRIPT language Programs 177

Using Journalling 177

Previewing POSTSCRIPT language Graphics 178

Talking Directly to the Server 178

A Sample Session 178

Connecting to Remote NeWS Servers 179

A.5. A Sample `psh` Program: `test.psh` 180

A.6. Dictionaries and the Server 183

Modifying the NeWS Server 183

`startup.ps` 183

`user.ps` 183

Notes on Modifications 183

Modifying Your ''Root'' Menu 184

Saving Keystrokes 184

Changing Defaults 185

A

Using NeWS

This chapter explains how to start up the NeWS server, and gives some very basic tips for getting started with NeWS and POSTSCRIPT language programming, including how to ''personalize'' your server.

Information on installing NeWS is in the *NeWS Installation Guide*. The examples in this section assume you have installed NeWS in `/usr/NeWS` or have mounted NeWS over the NFS on `/usr/NeWS`; substitute its actual location if different.

NOTE — *You can run NeWS from any directory, but it is easier to run* `psh` *programs (see below) if you have NeWS mounted on (or have symbolic links to)* `/usr/NeWS`. *Some of the demo programs cannot be run directly from a shell without modification if you do not have a* `/usr/NeWS/bin/psh`, *although they can be run from the menu.*

A.1. NeWS Environment Variables

The NeWS server needs to know where its subtree has been installed, so that it can find fonts, images, `*.ps` files, etc. Provide this information by:

```
paper% setenv NEWSHOME /usr/NeWS
```

The NeWS `bin` directory must be in the search path of the NeWS server process. It is also useful to put the demo directory into your path. So:

```
paper% set path=( /usr/NeWS/bin /usr/NeWS/demo $path )
```

Running NeWS on two displays is not supported.

If your machine has multiple displays available, and you want NeWS to use a framebuffer other that `/dev/fb` as its default framebuffer, you can:

```
paper% setenv FRAMEBUFFER /dev/cgtwo0
```

or whatever device corresponds to the other framebuffer.

You can set these NeWS environment variables permanently in your `.login` file (for C shell users) or in your `.profile` file (for Bourne shell users).

Which Server Binary?

There are three versions of the NeWS server in $NEWSHOME/bin, news_server010, news_server010.debug, and news_server The file news_server(1) is a symbolic link to news_server010, whi the MC68010 (Sun-2) version of the server. This will run on Sun-2 *and* Su machines, but if you have a Sun-3 you should run news_server020 for increased performance. Read *Various Versions of the NeWS Server* in the *N Installation Guide* for more information on the different binaries (including 4).

The Debugging Server Binary

The third version of the NeWS server found in $NEWSHOME/bin, news_server010.debug has been compiled with full debugging enab Customers who have a reproducible crash are encouraged to reproduce it u this server. Run **dbx** on the resulting core file and submit the output from t **where** and **dump** commands with the bug report.

A.2. Starting up NeWS

The NeWS server news_server can be started in two ways.

From outside suntools

```
paper% news_server
```

From within suntools using overview(1)

```
paper% overview -w news_server
```

This sends diagnostic output to the shelltool or cmdtool from whic started the server[16]. The NeWS server takes over the screen, and when x SunView1 tools "underneath" will redisplay.

Section A.3, *SunView1 Binary Compatibility with NeWS* discusses some of t limitations on using NeWS and SunView1 tools together.

Server Initialization

In each case, the NeWS server starts by reading and executing the POSTSCRI language code found in the file init.ps. The standard init.ps file others it executes define the desktop background, the root menus the sampl dow package, and other packages; see Chapter 4, *Extensibility through POSTSCRIPT Language Files* for more information on the contents of these

Whenever the server opens a file (including init.ps) it first tries to ope file in the current directory, and if this fails, it looks for it in ${NEWSHOME}/lib. Usually the server is searching for NeWS/*filenam*

A.3. SunView1 Binary Compatibility with NeWS

If you start up NeWS directly from the console it is possible to run unmodifi SunView1 (or SunWindows-based) binaries within NeWS. Windows put u SunView1 will appear to float over NeWS, including even NeWS menus, but overlap other SunView1 based windows. One way of looking at this is tha NeWS, rather than the suntools(1) program, manages the root window.

[16] NB: You must be sure that the FRAMEBUFFER environment variable used by the NeWS server ma -d argument used by suntools. Both of these values default to /dev/fb, so if one is changed, the must be also.

SunView1 windows appear to update the display simultaneously with NeWS canvases. SunView1 windows are surrounded by a white margin to avoid "glitches" when the cursor is moved between them and NeWS canvases.

NOTE *Running SunView1 tools under NeWS when the latter is started up from within* `suntools` *is* **not** *supported.*

It is possible to use NeWS and SunView1 applications simultaneously. However, you must make note of what version of the Sun OS you are running under. **NeWS is not supported on versions of the Sun OS prior to 3.2.** While running NeWS and a version of the OS with a release number equal to or greater than 3.2:

- Multiple SunView1 applications may run at the same time.
- SunView1 menus and prompts look fine, even over the NeWS window. However, NeWS menus appear to slide under SunView1 windows.
- SunView1 cross-hairs also work fine.
- Gfxsubwindow-based applications work fine.
- Very old versions (back to 1.1) of SunWindows-based applications work as well as they would under `suntools`.

If you are determined to use NeWS on releases prior to 3.2, then you should add the following to your `startup.ps` file:

```
UserProfile /UIModule /Default put
```

If you make this modification, you will be able to type at `psterm` (the terminal emulator), but more complex actions such as making selections and typing to `itemdemo` will not work.

Bugs in SunView1/NeWS Coexistence

This ability to run SunView1 programs from NeWS is handy, but not fully developed yet. As a result there are numerous bugs.

Inconveniences

- A color SunView1 application needs the cursor over its window in order to see the application's true colors.
- All of NeWS repaints when a SunView1 window's size or position changes.
- Annoying

  ```
  Window display lock broken after time limit exceeded \
      by pid nnn
  ```

  messages appear on the console. You can adjust the display lock timeout by modifying the kernel with `adb(1)`; see Section 7.5, *Kernel Tuning Options*, in the *SunView1 System Programmer's Guide*.
- If you run NeWS using `overview(1)` from within `suntools(1)` and you then run SunView1 applications inside NeWS, the colormap flashes when you move the cursor between NeWS and the SunView1 applications. This is one reason why this configuration is not supported.

Screen Damage

You will sometimes see cursor "splotches." This happens when you SunView1 application over the cursor that is on NeWS's part of screen. also occur when moving from a SunView1 part of the screen to a NeWS part screen. To cure this, move any SunView1 window to cause repair and choc 'Redisplay' from the frame menu of the affected SunView1 application.

The bottom scanlines of SunView1 windows get damaged by the NeWS serv a white line appears.

Input Mismatches

SunView1 sets up the kb(4S) keyboard driver in the kernel to deliver encoc events, while NeWS uses an unencoded keyboard. The NeWS server resets th board state to SunView1 encoding when the mouse moves into a SunView1 dow and when NeWS exits, so you shouldn't notice the difference. However you are debugging in NeWS and the NeWS server dies, NeWS doesn't catch the nals that make it reset the keyboard state, and you may be left with the keyb producing random characters in SunView1 windows. The program `kbd_m` switches the keyboard between the different modes; you can `rlogin` to yc machine and type **`kbd_mode -e`** to reset to SunView1 mode, or add the lowing to your `rootmenu` SunView1 rootmenu file:

```
"Reset Keyboard" kbd_mode -e
```

Since the keyboard state is changed when you move the mouse between Ne and SunView1, *don't hold any keys down* when you move the mouse from world to the other; the world you were in to begin with never sees the key g up, so it is confused about the keyboard's state when you reenter it. T r leave you in secondary selection mode. You should be able to clear this by pressing the [Stop] key twice (usually [L1]) in NeWS or SunView1.

If this does not reset the state of the function keys, you can `rlogin` to you machine and type `clear_functions`(1) to get SunView1's selection m ism out of a constant secondary selection mode.

NeWS on the Sun-3/110

On 3/110 models with the LC (`cgfour`) color monitor, if things go wrong the system seems to hang, the wrong plane group is probably being display To get out of this state, `rlogin` into your machine and type **`switcher`** -e **`1`** to display the overlay plane or **`switcher -e 0`** to display the color p group. You can avoid a common cause of this problem if you create a sym link from `/dev/fb` to `/dev/cgfour0`.

A.4. Learning NeWS

The best way to learn NeWS is to read the POSTSCRIPT language books, trying out sample POSTSCRIPT programs in NeWS as you go (see *Previewing POSTSCRIPT language Graphics* below for more information). Then start examining and modifying the NeWS demo programs and the server's `*.ps` files. The following sections give further information on these topics.

Putting A Message in a Window

To simply display a message in a window, use the `say(1)` program. This has many options, but its default action is to put up a message in a window; for example

```
paper% say -b"Text Using Say" Hello There
```

will create a window, give it the frame header ''Text Using Say'', and display ''Hello There'' in it. Since NeWS is a network service, you can easily put up a message on a remote machine with this command; see Section A.4.3, *Connecting to Remote NeWS Servers*

The `psh` Command

The `psh(1)` command provides the easiest way to send POSTSCRIPT programs to the NeWS server. There is a manual page for it in `$NEWSHOME/man`, which is also printed in the back of this manual.

The `psh` command establishes a connection to the NeWS server and sends it POSTSCRIPT language fragments. If your program can live directly in the NeWS server, (i.e. it doesn't have to communicate with a C client side), you can use `psh` to run the code in the server.

Running POSTSCRIPT language Programs

The NeWS program you send to the server can create its own window and even define its own menu and input handling. The sample program described below in Section A.5, *A Sample* `psh` *Program:* `test.psh` uses this method, and it's how many of the graphics demo programs work, such as `bounce`, `colorcube`, `itemdemo`, etc.

`psh` need not create any window at all. For example, you might want to change the visual feedback that occurs when you move a window around the screen; this can be changed with the 'User Interface => Window Management Style => Flip Drag' menu item, and to do the same from the command line, the following code would suffice:

```
echo '/dragframe? dragframe? not store' | psh
```

Using Journalling

When you select 'Applications/ ⇒/ Journal' from the root menu, a new pull-right menu is added to the root menu.

From this you can start recording user input events, stop recording, play them back, or remove journalling. You can also bring up a control panel with buttons for controlling journalling, the speed of playback, auto-repeat, the journalling file to use, etc. See Ch.4, *Extensibility through POSTSCRIPT language Files* for more detailed information.

Previewing POSTSCRIPT language Graphics

You can use psh to directly preview graphics by creating your own wi[illegible] defining a **PaintClient** procedure for it that includes your POSTSCRIPT lang code. **PaintClient** is called whenever the window is resized or damaged (s Section 7.4, *Window Methods*).

However, you must be aware of some differences between psh and a POSTSCRIPT language printer. Firstly, various printer-related commands su **showpage** are meaningless or pseudo-implemented in the NeWS window en ment. Secondly, the default Sun LaserWriter coordinate scheme is one unit one point on the paper (a point is 1/72 of an inch), so that a U.S. letter-sizec goes from (0,0) to (612,792), whereas the window canvas is scaled in pixels begin with.

The program psview(1) implements a page previewer that gets around the problem by including some additional code to scale the coordinate space to match the canvas.

Talking Directly to the Server

The NeWS server is a POSTSCRIPT language interpreter, and you can use it ir tively to program and debug.[17] Usually you use the psh command to conn the server, then run the **executive** command to start an *executive*, an interac session with the server.

```
paper% psh
executive
Welcome to NeWS Version 1.1
```

A Sample Session

Once running an executive, you can type in any POSTSCRIPT language com mands you want. The following miscellany perform some arithmetic, defin function called 'centigrade', and find out some stuff about the NeWS server further commands print some words and draw arcs of various styles on the screen.

[17] (see Chapter 8, *Debugging*).

```
340 1024 mul =
348160
/centigrade {
    32 sub 5 mul 9 div
} def
70 centigrade =
21.1111
32 centigrade =
0
currentcanvas =
canvas(width, height, root)
100 100 moveto
/Times-Italic findfont 24 scalefont setfont
(Hello world!) show

newpath
150 200 50 90 0 arc
stroke

1 setlinequality  1 setlinejoin  30 setlinewidth
300 200 50 90 0 arc
stroke
```

Connecting to Remote NeWS Servers

NeWS is a network-based window system, so you can connect to remote NeWS servers and display output on them (remember, the *server* runs on the machine with the display and keyboard, providing them as a resource for the *client* program).

The environment variable NEWSSERVER determines which server client programs will access; by default they access the local host. There is a utility program, setnewshost(1) which outputs the correct setting of the NEWSSERVER variable for a given remote host. You can also craft the value of NEWSSERVER yourself using the information about its format in Section 15.1, *Contacting the Server*.

After you define NEWSSERVER, say and other NeWS client programs will display their output on *remote_host*, and psh with no arguments will connect to *remote_host* allowing you to run an interactive programming session on a remote machine. For example, to display a message on machine neighbor:

```
paper% setenv NEWSSERVER `setnewshost neighbor`
paper% say -bLunch -c -w -100,200 "Come have lunch" &
```

neighbor has to have allowed a NeWS connection from paper for this all to work. See Chapter 11, *NeWS Type Extensions*, section 11.9, *NeWS Security* for more information.

More commonly, you can use the network aspect of NeWS in reverse to run NeWS applications on fast remote machines while you interact with them on your own workstation. For example, you could use the on(1) remote execution service

(which preserves environment variables like NEWSSERVER) from with psterm(1) terminal emulator (which explicitly sets NEWSSERVER to the N server it is running on) to run the roundclock program to determine the t on a remote machine.

A.5. A Sample psh Program: test.psh

The following program is a script that creates its own window and displays of several graphic patterns, depending on the menu item selected. The sourc in $NEWSHOME/clientsrc/client/test.psh.

The sample program uses psh(1) to connect to the NeWS server. The first li

```
#! /usr/NeWS/bin/psh
```

means that if the file is executable, it will be run by /usr/NeWS/bin/ps you do not have NeWS mounted on /usr/NeWS, you have several choices:

1. Change the first line to reflect the location of psh on your host.
2. Type

```
paper% psh wherever/test.psh
```

   to run the script.

3. Modify the script to be:

```
#! /bin/sh
psh << \EOF
... current script
EOF
```

   (the <<\EOF on the command line is a shell convention indicating that entire file up to the string EOF should be sent to the standard input of p the backslash tells the shell not to perform metacharacter substitution o input, which is useful if the POSTSCRIPT language code contains shell r characters).

When reading POSTSCRIPT language source, it's often easiest to start from the bottom and work backwards, since subprocedures are usually defined before the main body of the code

The patterns are drawn by the procedures 'Lines,' 'Circles,' 'Rects' and 'T these are defined first. Then a 'main' procedure is defined: this creates the w dow and a menu for it using procedures from the *LiteWindow* and *LiteMenu* classes, which handle event processing (putting up the menu when the menu mouse button is pressed, calling the 'Draw' procedure to repaint, etc.). 'ma called to begin the program.

The procedure 'Draw' is set to one of the graphics procedures to begin with is reset when one of the procedures is selected from the menu. Note that if ' is selected, 'Draw' is set to all four graphics procedures, so each is rapidly c in turn. The window's **PaintClient** procedure is set to 'Draw', so wheneve window is damaged or redisplayed, the current graphics procedure is called.

```
#! /usr/NeWS/bin/psh
% define each of the four possible drawing routines
% 'Lines', 'Circles', 'Rects', and 'Text'.
% Note the use of pause in each drawing routine. This allows other programs
% to run simultaneously.
```

```
/Lines {
gsave                                                   % It's a good idea to do a gsave...
                                                        % grestore around graphics operations.
  1 fillcanvas clippath pathbbox scale translate
  .1 .1 1 {0 0 moveto dup 1 lineto 0 0 moveto 1 exch lineto pause} for
  0 setgray stroke
grestore
} def
/Circles {
gsave
  1 fillcanvas clippath pathbbox scale translate
  0 .1 .4 {dup 0 0 1 1 insetrect ovalpath 2 mul setgray fill pause} for
grestore
} def
/Rects {
gsave
  1 fillcanvas clippath pathbbox scale translate
  0 .1 .4 {dup 0 0 1 1 insetrect rectpath 2 mul setgray fill pause} for
grestore
} def
/Text {
gsave
  1 fillcanvas 0 setgray
  /Fonts [                                              % This is an array of font names
    (Times-Roman) (Times-Bold) (Times-Italic)
    (Helvetica) (Helvetica-Bold) (Helvetica-Oblique)
    (Courier) (Courier-Bold) (Courier-Oblique)
    (Symbol) (Boston) (Cyrillic)
  ] def
  /PointSize 24 def
  /y 10 def
  Fonts {                                               % The forall operator below performs
                                                        % this procedure for each font in the
                                                        % 'Fonts' array
    dup findfont PointSize scalefont setfont 10 y moveto
    10 {dup show 15 0 rmoveto} repeat pop
    /y y PointSize 1.1 mul add def
    pause
  } forall
grestore
} def
/Draw {Text} def                                        % Initial drawing procedure is 'Text'
% 'Text' has the window draw me so that I inherit certain
% side effects, such as forking the PaintClient procedure
% and setting the graphics state.
/CallDraw {/paintclient win send} def

/main {
  /win framebuffer /new DefaultWindow send def          % Create a window
  {                                                     % Modify the window. There are default
                                                        % procedures for each of these.
    /PaintClient {Draw} def                             % /PaintClient will be called
                                                        % whenever my image needs to be
```

```
                                                             % repaired or redisplayed.
    /FrameLabel (Demos!) def
    /IconImage /hello_world def
    /ClientMenu [                                            % Make the menu and give it to the
                                                             % window event manager to handle.
      (Lines)        {/Draw {Lines} store CallDraw}
      (Ovals)        {/Draw {Circles} store CallDraw}
      (Rectangles)   {/Draw {Rects} store CallDraw}
      (Text)         {/Draw {Text} store CallDraw}
      (All!)         {/Draw {Lines Circles Rects Text} store CallDraw}
    ] /new DefaultMenu send def                              % /new is one of the well-known
                                                             % methods in the menu class.
  } win send                                                 % Sending a procedure to an object
                                                             % will cause it to be executed in
                                                             % the object's context.
  /reshapefromuser win send                                  % Ask the user to shape the window.

  % Do initialization. Then have my window mapped (made visible)
  % which will cause the paint procedure to be called.

  /map win send                                              % Map (& activate) the window.
                                                             % Damage will cause PaintClient
                                                             % to be called.

} def

main                                                         % start everything going
```

A.6. Dictionaries and the Server

When the NeWS server starts up, it runs a file called `init.ps`. `init.ps` in turn loads several POSTSCRIPT language files that implement a variety of packages; see Section 4.2, *File Organization* for a summary of their organization and contents. These are all ASCII POSTSCRIPT language files that you can and should look at.

Modifying the NeWS Server

It is possible to modify these files, or make copies of them and put them in the directory from which you start the NeWS server. Thus you can override the default `init.ps` (or any of the files it loads) by putting a private version of the file in the directory from which you start the server; For example, if you create a `NeWS` directory in the directory from which you start `news_server`, and put your own `init.ps` file in that, the server would run your version instead. However, it is better to override the default procedures specified in these start up files by creating your own `startup.ps` and `user.ps` files. These aren't supplied with NeWS; users who want to change the server's behavior are encouraged to create their own.

You should place `startup.ps` and `user.ps` in your home directory or the directory from which you start NeWS, (*not* the directory the NeWS server itself resides in). `init.ps` first looks in the directory the NeWS server was started from, then in your home directory (given by the `HOME` environment variable) when it tries to load `startup.ps` and `user.ps`.

`startup.ps`

Before it loads anything else, the default `init.ps` file looks to see if a file called `startup.ps` exists. If it does, it executes the POSTSCRIPT language commands therein. These commands would typically be to set flags, such as **verbose?**, or to install a special **PaintRoot** procedure that is used during startup.

NOTE *Since the system's own POSTSCRIPT language files have not been read in when* `startup.ps` *is loaded, you cannot use any of the routines described in Chapter 4,* Extensibility through POSTSCRIPT Language Files, *or the packages (windows, cursors, etc.) they implement.*

Then `init.ps` loads a standard set of POSTSCRIPT language files that define the classes, packages and user interface for NeWS.

`user.ps`

Next, `init.ps` looks to see if a file called `user.ps` exists. If it does, it executes that too. `user.ps` is a convenient place for you to override default settings in the standard `init.ps`, and to define useful procedures for your own use.

Then the server starts listening for connections and for mouse and keyboard events.

Notes on Modifications

When client programs are run and first start defining procedures, they make entries in a per-process user dictionary. However, the `user.ps` file adds procedures to the system dictionary, which has a finite amount of room available. More importantly, the system dictionary is shared by all processes, so you do not want to clutter it up with lots of definitions because this will increase the risk of name clashes.

If you are going to define lots of new functions, it's best to create your own dictionary from within `user.ps` as follows :

```
% Define my VDI emulation routines
systemdict /myVDIdict known not {
    systemdict /myVDIdict 50 dict put    % size to however many entries are in it
    myVDIdict begin
    /VDIrange 34200 def
    etc.
    end
} if
```

This checks to see if 'VDIdict' is already defined in the system dictionary; if it isn't, it creates its own dictionary, only adding one entry to the system dictionary. You can access your own dictionary of extensions as follows:

```
myVDIdict begin
VDIrange 4 mul
etc.
end
```

(Or you can use the **get**, **store** and **put** primitives.)

Here are some examples of modifications you can make in `user.ps`.

Modifying Your "Root" Menu

The following code in `user.ps` will add a new pullright menu, *myStuff*, to the root menu. This in turn has two items, another pullright menu called *myWork* and a menu item that runs the UNIX program *mygo*.

```
/projectmenu [
    (test program)    { (work.tst) forkunix }
] /new DefaultMenu send def

/mymenu [
    (myWork =>)       projectmenu
    (my Go game)      { (mygo) forkunix }
] /new DefaultMenu send def

0 (myStuff =>) mymenu /insertitem rootmenu send
```

Saving Keystrokes

If you often connect to the server directly, you can redefine commonly used commands to save typing.

You should not use these shortened names in client programs since the definitions will not exist on other machines.

The following code added to your `user.ps` redefines several common commands to save keystrokes.

```
% Some aliases
/ps {pstack} def
/cds {countdictstack =} def
/fb framebuffer def
% Debugger
/dbe {dbgbreakenter} def
/dbx {dbgbreakexit} def
/dc {dbgcontinue} def
/dlb {dbglistbreaks} def
/dwb {dbgwherebreak} def
(keystroke savers ps, cds, fb, dbe, dbx, dc, dlb and dwb defined\n) print
```

Changing Defaults

The following code added to your `user.ps` will load the optional debug package and change item dragging behavior if running on a color display so that only a wire frame of the window is dragged, not the entire window which requires more memory to perform this adequately.

```
(Loading debug.ps\n) print     (debug.ps) run
```

B

Class *LiteItem*

Class *LiteItem* **189**

B.1. Class **Item** 189
B.2. Two Sample Items 191
Sample Items Test Program 194
B.3. Class **LabeledItem** 196
B.4. Subclasses of **LabeledItem** 198
B.5. **LabeledItem** Subclass Details 200

B

Class *LiteItem*

This chapter presents a class-based items package, called ''LiteItem.'' Items are simple, graphical input controls, like *SunView*'s panel items. The item package is a further demonstration of the use of classes and packages besides the window and menu packages described in Chapter 7, *Window and Menu Packages*. The item package is only used by the `itemdemo` demo program. The POSTSCRIPT language code for the items package is in the file `liteitem.ps` in `$NEWSHOME/lib/NeWS`.

CAUTION **This package is included for demonstration purposes only. No support for it in the future is implied.**

The item package currently implements the base class, **Item** (which is useless by itself), the subclass **LabeledItem** (which also is useless), and several practical subclasses of **LabeledItem** (which *are* useful).

B.1. Class Item

A common need in interactive systems is a simple, user-definable, graphic, interactive, input/output object. Examples are buttons, sliders, scrollbars, dials, text fields, message areas, and the like. The class **Item** defines a skeleton for such an object.

An item has these major components:

- A canvas that depicts the item and is the target of the item's input.
- A set of procedures that paint the canvas and handle activation and tracking events.
- A current value and a procedure that notifies the client when that value changes due to action of the tracking procedures.
- Methods for creating, moving and painting the item, and for returning the item's location and bounding box.

There are two utilities for items that reside outside of the class itself: **forkitems** and **paintitems**:

forkitems

items **forkitems** process
Takes an array or dictionary of items and launches a process looking for an activation event (generally a mouse down event) for each of the items. Wh this occurs, a second event manager is forked that looks for events this parti item is interested in.

paintitems

items **paintitems** –
Sends the **/paint** message to each of the items in an array or dictionary of it

Let's take a look at the definition of class **Item**:

```
/Item Object [
% instance variables
    /ItemWidth                          % item's width,
    /ItemHeight                         % ...and height,
    /ItemParent                         % ...and parent canvas (from new)
    /ItemCanvas                         % the canvas we created for the item
    /ItemValue                          % the canvas' current value
    /ItemInitialValue                   % the value it started out with
    /ItemPaintedValue                   % the value it currently shows
    /StartInterest                      % the interest which activates the item
    /ItemInterests                      % interests used to track item
    /ItemEventMgr                       % ...the tracking process
    /NotifyUser                         % the user's notify proc
] classbegin
% default variables
    /ItemFont        DefaultFont def    % the item's font
    /ItemTextColor   0 0 0 rgbcolor def % ...& text color
    /ItemBorderColor ItemTextColor def  % ...& border color
    /ItemFillColor   1 1 1 rgbcolor def % ...& background color
% class variables; mainly the std client procs
    /PaintItem   nullproc def           % the core of the /paint method
    /ClientDown  nullproc def           % procedures installed in
    /ClientDrag  nullproc def           %   the activated (tracking)
    /ClientEnter nullproc def           %   process
    /ClientExit  nullproc def
    /ClientKeys  nullproc def
    /ClientUp    nullproc def
    /StopOnUp?   true def               % deactivate on up event?
% methods
    /new                                % parentcanvas width height => inst
    /makecanvas                         % - => -
    /makeinterests                      % - => -
    /move                               % x y => - (Moves item to x y)
    /moveinteractive                    % item's backgroundcolor => -
                                        % (interactively moves the item)
    /paint                              % - => - ([Re]paints item)
    /location                           % - => x y
    /bbox                               % - => x y w h
classend def
```

The canvas and its "looks" are defined by:

> **ItemWidth, ItemHeight, ItemParent, ItemCanvas, ItemFont, ItemTextColor, ItemBorderColor, ItemFillColor**

The parent canvas, height, and width are specified by the **/new** method. The others are initialized by the class and may be changed by the programmer or user.

The set of procedures for painting the canvas and handling activation and tracking events are:

> **PaintItem, StartInterest, ItemInterests, ItemEventMgr ClientDown, ClientDrag, ClientEnter, ClientExit, ClientKeys, ClientUp, StopOnUp?**

The **PaintItem** procedure is called by the **/paintitem** method, after it sets the canvas and does some minor bookkeeping. **StartInterest** is an event used by the *forkitems* utility to determine when to fork a second event manager, **ItemEventMgr**, to perform ''tracking'' of the item. **StartInterest** defaults to a mouse down event and generally is not overridden. **ItemInterests** is a dictionary of events used to track the item. It defaults to a set of events determined by which of the 'ClientFoo' procedures have been overridden to be non-null. It is made by the **/makeinterests** method, which is generally invoked by the **/move** method as part of the item's deferred initialization. Clients are free to call **ItemInterests** themselves, however, if the need arises. **ItemEventMgr** is the tracking process and is null when tracking is not being performed. **StopOnUp?** is a boolean (default = true) that tells the tracking process whether to terminate on an up mouse event. (At present, only text items do not terminate on an up mouse event.)

The current value and notification procedures use:

> **ItemValue, ItemInitialValue, ItemPaintedValue, NotifyUser**

ItemValue is the current value of the item. For example, it might be the string currently in a type-in item or true/false for a button item (indicating whether the button is currently on or off). **ItemInitialValue** is set to **ItemValue** when the item is activated for tracking. **ItemPaintedValue** is set to the value currently painted. These last two values are used to maintain a simple state machine by class implementations. **NotifyUser** is a procedure used to alert the client of changes in state.

B.2. Two Sample Items

These two subclasses of **Item** implement a simple toggle button and a simple slider. They both override the **/new** method, adding the initial **ItemValue** and the **NotifyUser** procedure to the argument list. Notice the way overriding is done:

```
/new super send begin
  ...
  currentdict
end
```

This is a standard POSTSCRIPT language programming style.

'SampleToggle' provides tracking by implementing the client 'Down', 'Up', 'Enter', and 'Exit' procedures. **ItemValue** is treated as a boolean, with true meaning ''on''. 'Down' and 'Enter' simply assign **not ItemInitialValue** to **ItemValue**, while 'Exit' resets it to **ItemInitialValue**. 'Up' simply calls the 'notify' procedure if the state has changed. 'SampleToggle' adds no instance or class variables.

Here are two toggles, one on and the other off, and the implementation [illegible] class 'SampleToggle':

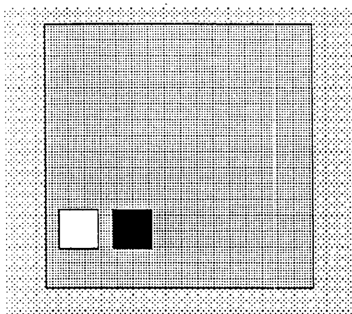


Figure B-1 *Two Instances of Class* 'SampleToggle'

```
/SampleToggle Item []
classbegin
   /new {                              % initialvalue notifyproc parent width height =:
      /new super send begin
         /NotifyUser exch cvx def
         /ItemValue exch def
         currentdict
      end
   } def

   /PaintItem {
      ItemValue
         {0 fillcanvas}
         {1 fillcanvas 0 strokecanvas} ifelse
   } def
   /ClientDown {ItemInitialValue not SetToggleValue} def
   /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
   /ClientEnter {ClientDown} def
   /ClientExit {ItemInitialValue SetToggleValue} def

   /SetToggleValue {                   % value => – (set value & paint toggle)
      /ItemValue exch store
      /paint self send
   } def
classend def
```

The 'SampleSlider' provides tracking by implementing the client 'Down 'Up', and 'Drag' procedures. The 'Down' and 'Drag' procedures are ider simply projecting the current *x* coordinate of the mouse onto the slider.

Here is a slider and its implementation:

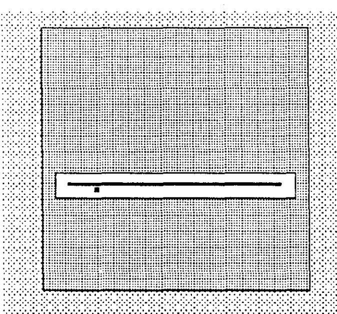


Figure B-2 *An Instance of Class* 'SampleSlider'

```
/SampleSlider Item [/SliderX /SliderY /SliderWidth /SliderHeight]
classbegin
  /new {                          % initialvalue notifyproc parent width height => item
    /new super send begin
      /NotifyUser exch cvx def /ItemValue exch def
      /SliderX       ItemHeight 2 div 1 sub def
      /SliderY       ItemHeight 2 div def
      /SliderWidth     ItemWidth ItemHeight sub def
      /SliderHeight    2 def
      currentdict
    end
  } def
  /PaintItem {
    ItemCanvas setcanvas 1 fillcanvas 0 strokecanvas
    SliderX SliderY SliderWidth SliderHeight rectpath fill
    ItemValue 0 PaintSliderValue
  } def
  /ClientDown {
    SetSliderValue
    ItemValue ItemPaintedValue ne {
      ItemPaintedValue 1 PaintSliderValue
      ItemValue      0 PaintSliderValue
    } if
  } def
  /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
  /ClientDrag {ClientDown} def
  /PaintSliderValue {             % value gray => –
    setgray
    SliderX add SliderY 5 sub 4 4 rectpath fill
    /ItemPaintedValue ItemValue store
  } def
  /SetSliderValue {
    /ItemValue
      CurrentEvent geteventlocation pop SliderX sub
      0 max SliderWidth min store
  } def
classend def
```

Sample Items Test Program

Now we'll play with these procedures using a simple test program. Th[illegible] procedure simply prints the value of the item using the **printf** utility. We's building a canvas and painting it with 'itembackground.' Then we make items, a button and a slider, putting them in a dictionary called 'items.' W paint them and fork an activation event manager.

This is usually all you need; we throw in an extra event manager, 'p1,' that the middle mouse button to move the items interactively with the 'slideiter procedure. It then prints out the new location. (This is a poor man's item e that often is useful.)

Here's what the test looks like, and its implementation:

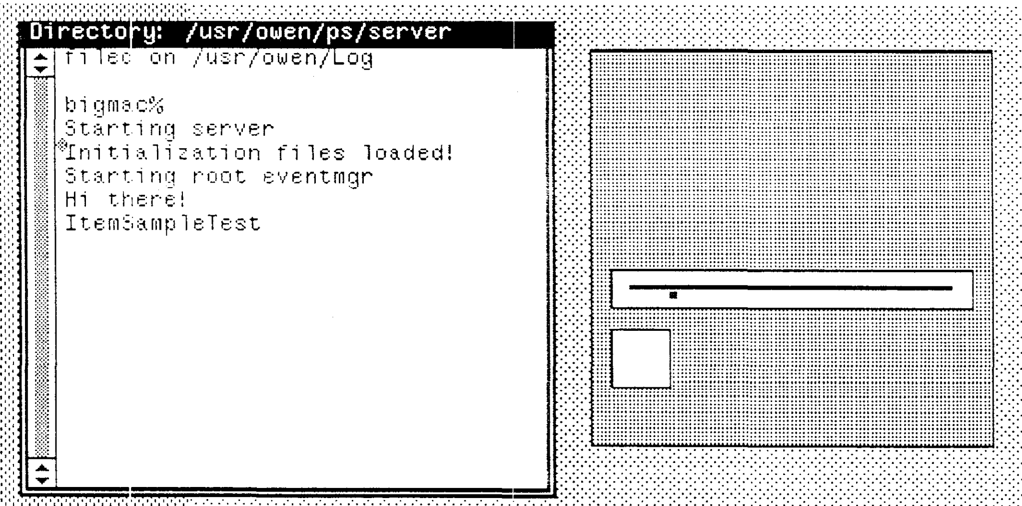


Figure B-3 *The Sample Test Program*

```
/ItemSampleTest {
   /notify {ItemValue (ItemValue: % \n) printf} def
   /itembackground .75 def
   /can framebuffer 200 200 createcanvas def
   can setcanvas 200 100 movecanvas currentcanvas mapcanvas
   itembackground fillcanvas 0 strokecanvas

   /items 10 dict dup begin
      /sampletoggle
         false /notify can 30 30 /new SampleToggle send def
         10 30 /move sampletoggle send
      /sampleslider
         20 /notify can 180 20 /new SampleSlider send def
         10 70 /move sampleslider send
   end def
   items paintitems /p items forkitems def

   /slideitem {                                        % - (event) => -
      /item CurrentEvent /action get def
      items itembackground /moveinteractive item send
      (New bbox: % % % % \n) [/bbox item send] printf
   } def
   /p1 [
      items {                                          % key item
         /item exch def pop                            % -
         MiddleMouseButton {slideitem}                 % but proc
         1 dict dup DownTransition item put            % but proc dict
         item /ItemCanvas get eventmgrinterest
      } forall
   ] forkeventmgr def
} def
```

After pushing the toggle and sliding the slider, we have:

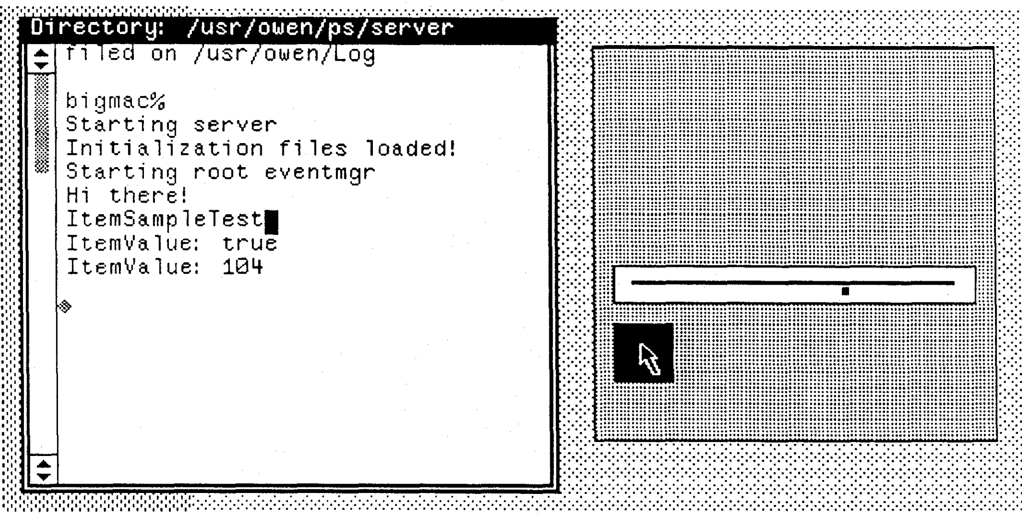


Figure B-4 *Use of the Sample Test Program*

After we push the middle button on the slider and move it, we get:

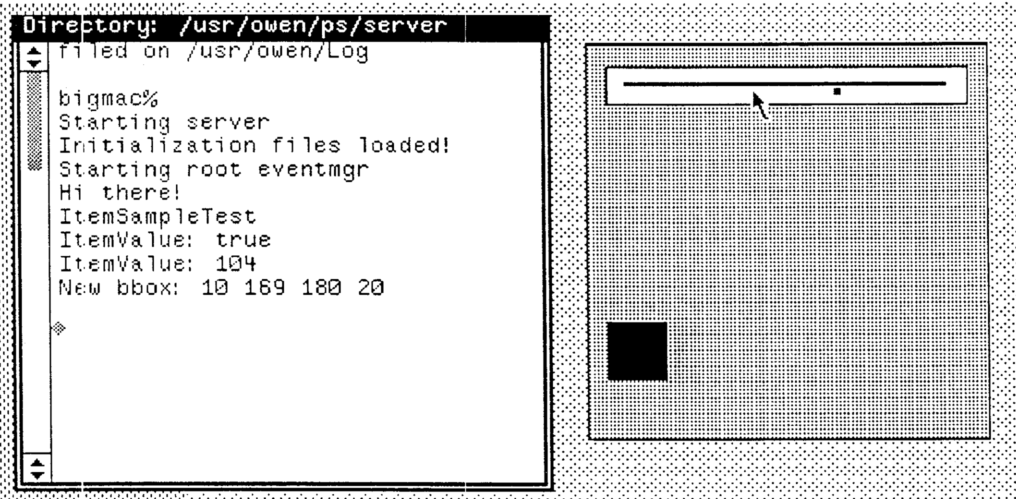


Figure B-5 *Use of the Sample Test Program — Moving the Slider*

What to notice here is the simplicity and power both of the program and sa items, and of the NeWS programming environment. The implementation of items and test program, and the testing of them both in the interpretive NeW environment takes *very* little time.

B.3. Class LabeledItem

Most items are more elaborate than the preceding examples. **Class Labele** implements a more common item; one that has a label-object pair, and an optional frame. The (abbreviated) class definition is:

```
/LabeledItem Item
dictbegin
% instance variables
   /ItemObject    nullstring def        % The item's "object"
   /ObjectX       0 def                 % and bounding rect:
   /ObjectY       0 def
   /ObjectWidth   0 def
   /ObjectHeight  0 def
   /ItemLabel     nullstring def        % The item's "label"
   /LabelX        0 def                 % and bounding rect:
   /LabelY        0 def
   /LabelWidth    0 def
   /LabelHeight   0 def
   /ItemBorder    2 def                 % Extra space around the item
   /ObjectLoc     null def              % Label-Object position
   /ItemGap       5 def                 % Distance between object & label
   /ItemFrame     0 def                 % Draw frame if not zero
   /ItemRadius    0 def                 % Radius of frame
dictend
classbegin
% default variables
   /ItemLabelFont  Item /ItemFont get def
% class variable: over-ride of PaintItem
   /PaintItem                           % – => –
% methods: over-ride new
   /new                                 % label obj loc notify parent width height => instance
% utilities used to manipulate label-object pair
   /LabelSize                           % – => width height
   /ShowLabel                           % – => –
   /ShowObject                          % – => –
   /EraseObject                         % – => –
   /AdjustItemSize                      % – => –
   /CalcObj&LabelXY                     % – => –
classend def
```

The label and object and their item-relative bounding rectangles are defined by:

> **ItemLabel, LabelX, LabelY, LabelWidth, LabelHeight, ItemObject, ObjectX, ObjectY, ObjectWidth, ObjectHeight**

The **ItemLabel** and **ItemObject** are either a string, an icon keyword, or a procedure keyword. If it is a procedure, it takes a boolean as an argument: true causes it to draw itself; false causes it to return its width and height. The **ItemLabelFont** is bound to the label, the **ItemFont** is bound to the object.

The item's layout metrics are defined by:

> **ItemBorder, ObjectLoc, ItemGap, ItemFrame, ItemRadius**

ItemBorder is the space between the item bounding box and its label-object pair. **ObjectLoc** is the position of the object relative to the label. It may be any of **/Right**, **/Left**, **/Top**, **/Bottom**. **ItemGap** is the space between the label-object pair. **ItemFrame** is the size of the frame to draw around the item. It should be no greater than **ItemBorder**. **ItemRadius** is the curvature of the item's border. Zero represents a rectangular shape; a number between 0 and .5 represents a rounded rectangle whose radius is that fraction of the shortest edge; and any

other number is used as the absolute curvature of the rounded rectangl

The two overrides are the /**new** method and the /**PaintItem** procedure calle the /**paint** method. Note that /**new** adds **label**, **obj**, **loc**, and **notify** to the a ments of its superclass. These are bound to **ItemLabel**, **ItemObject**, **ObjectLoc**, and **NotifyUser**, respectively.

Class **LabeledItem** contains a few utilities that are used by its subclasses:

> **LabelSize**, **ShowLabel**, **ShowObject**, **EraseObject**, **AdjustItemSize** **CalcObj&LabelXY**

LabelSize returns the height and width of the label. **ShowLabel** and **Shov ject** paint the label and object in the item's canvas; **EraseObject** erases th object. **AdjustItemSize** and **CalcObj&LabelXY** are two layout utilities. **AdjustItemSize** is used to insure the item is large enough for its label-obje while **CalcObj&LabelXY** is used to adjust the label-object pair's relative tions. Note that **CalcObj&LabelXY** adds the initial values of the label an object locations, which default to 0, to the calculated locations, thereby prc for slight adjustments by the programmer.

B.4. Subclasses of LabeledItem

This section presents several practical subclasses of **Class LabeledItem**, s ing how they are used by client programs. There are further examples of i usage in the `itemdemo` program provided in the standard release. The i mentation of these classes is included in the `item.ps` file that implemen **Class Item** and **Class LabeledItem**. Programmers wanting to implement own items should look at these implementations; they are generally l n page of POSTSCRIPT language.

The subclasses are:

- **ButtonItem**: provides a simple activation/confirmation item
- **CycleItem**: provides check boxes and choices
- **SliderItem**: provides a continuous range of values
- **TextItem**: provides a type-in area
- **MessageItem**: provides an output area
- **ArrayItem**: provides an array of choices

This window contains one of each of these items:

Items

ButtonItem! CycleItem One Two Four Six ArrayItem MessageItem ItemValue: Foo

TextItem: Foo

SliderItem: 0

Figure B-6 *Subclasses of* **LabeledItems**

Things to notice:

- All items except button items have the following arguments:

  ```
  label object location notifyproc parentcanvas width height
  ```

  Buttons have no object; thus leave out **object** and **location**. Cycle and array items have multiple objects in an array.

- All four object locations are visible in the sample:
  - The text and slider items use /**Right**
  - The cycle item uses /**Left**
  - The message item uses /**Top**
  - The array item uses /**Bottom**

- The **width** and **height** parameters are hints only; the **AdjustItemSize** procedure will increase these if necessary. The minimal size will be enough to contain the label-object pair separated by *ItemGap* with a border of **ItemBorder**. The message item in the example below uses this by using a 0,0 size for the canvas but a large empty string as the initial value of the message.

- The values calculated by **CalcObj&LabelXY**

  ```
  LabelX LabelY ObjectX ObjectY
  ```

  can be adjusted by assigning initial values to any of them. This is used in the check box example above:

  ```
  /cycle (CycleItem) [/panel_check_off /panel_check_on]
     /Left /notify can 0 0 /new CycleItem send
     dup /LabelY -4 put 10 70 /move 3 index send def
  ```

- The label or object can generally be either a string, an icon name, or a procedure. The procedure takes a boolean which indicates whether to draw the object (true) or to return its width and height (false). The array object above shows both an icon and a drawing procedure (the box):

```
/drawing {
   {ItemBorderColor setcolor 1 1 14 14 rectpath stroke}
   {16 16} ifelse
} def
```

Use of these items follows the general pattern:

- A canvas is created for containing the items. This is generally done b creating a window and getting its **ClientCanvas**. The window's **PaintClient** procedure should include a call to **paintitems**.
- A dictionary (or array) of items is created using the **/new** message to t item class of interest followed by a **/move** message to this instance.
- This collection of items is activated by calling **forkitems**. The items' notification procedures will be used to perform the activities the progr desires.

Here is a minimal example of this style. We create a window with a butto message item. The button's notify procedure simply prints "Button Press the message item.

```
/win {
   /PaintClient {items paintitems} def
} makewindowfromuser def
/can win /ClientCanvas get def
/notify {(Button Pressed!) /print items /message get send} def

/items 50 dict dup begin
   /message (Message:) (                    )
      /Right {} can 0 0 /new MessageItem send
      10 10 /move 3 index send def
   /button (Button) /notify can 0 0 /new ButtonItem send
      10 30 /move 3 index send def
end def
/p items forkitems def
```

This is the resulting window:

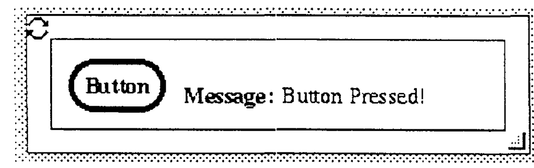


Figure B-7 *Typical Item Usage*

B.5. LabeledItem Subclass Details

This section presents details on the use of the **LabeledItem** subclasses.

- **ButtonItem**: **ItemValue** is a boolean; true if pressed, false if not. Ge the **ItemValue** is never used; the **NotifyUser** is called to perform the button's activity. **ButtonItems** differ from the rest of the **LabeledIte** not having an object and object location in the arguments to */new*.

- **CycleItem**: **ItemObject** is an array of objects. **ItemValue** is the index of the currently displayed object in that array. The cycle starts at zero and progresses one each ''push'' of the item. **NotifyUser** is called when the cycle changes value.

- **SliderItem**: The object is an array consisting of three integers: the minimum value, the maximum value, and the initial value for the slider. The **ItemValue** is the current value of the slider, and **NotifyUser** is called when the button is released. You can be notified continuously by overriding **ClientDrag**:

```
/ClientDrag {/ClientDrag super send NotifyUser} def
```

- **TextItem**: **ItemValue** is the current string being displayed. The object is the initial string. **NotifyUser** is called whenever there is any change to **ItemValue**. Text items differ from the others in the way they use the mouse. **MouseDown** activates the text item if it is not yet active, and changes the caret location if it already active. The item is de-activated by activating another text item or by exiting the parent canvas of the text item. Keyboard motion is available in the text item using standard Emacs control sequences.

- **MessageItem**: The object in a message is its initial value. This need not be text! Message items have two additional methods: **/print** and **/printf**. **/print** takes a single argument, generally a string, and displays it as the item's new object. **/printf** has two arguments: a format string and an argument array. See the previous chapter for sample usage. **NotifyUser** is called whenever a new message is posted. It should generally be an empty procedure. **ItemValue** is the current message.

- **ArrayItem**: The object is an array of equal length arrays. The ''inner'' arrays are the subsequent rows. The sample array item was created by:

```
(ArrayItem) [
   [(One) (Two) /panel_text]
   [(Four) /drawing (Six)]
] /Top /notify can 0 0 /new ArrayItem send
```

  ItemValue is an array of indices of the current selection and is initialized to [0 0]. **NotifyUser** is called from **ClientUp** if the **ItemValue** changed from its initial value.

C

NeWS Operators

NeWS Operators **205**

C.1. NeWS Operators, Alphabetically 205
C.2. NeWS Operators, by Type 207

C

NeWS Operators

This appendix lists all the current NeWS operators, alphabetically first, then by type.

C.1. NeWS Operators, Alphabetically

listenfile	**acceptconnection** file	listens for connection
num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
num	**arctan** num	computes arc tangent
–	**awaitevent** event	blocks for *event*
num	**blockinputqueue** –	block input events
–	**breakpoint** –	suspends current process
w h bits matrix proc	**buildimage** canvas	constructs canvas object
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clipcanvas** –	clip to canvas boundary
–	**clipcanvaspath** –	set current path to clip
process	**continueprocess** –	restart suspended process
color	**contrastswithcurrent** boolean	compare colors
dx dy	**copyarea** –	copy current path to *dx, dy*
–	**countinputqueue** num	returns count of input queue
string	**createdevice** canvas	create new canvas
–	**createevent** event	create an event
–	**createmonitor** monitor	create monitor object
canvas	**createoverlay** canvas	create overlay canvas
–	**currentautobind** boolean	autobinding enabled?
–	**currentcanvas** canvas	current canvas
–	**currentcolor** color	current color
–	**currentcursorlocation** x y	returns mouse coordinates
–	**currentlinequality** n	current line quality
–	**currentpath** shape	return current path
–	**currentprintermatch** boolean	return **printermatch** value
–	**currentprocess** process	return current process object
–	**currentrasteropcode** num	rasterop combination function
–	**currentstate** state	returns **graphicsstate** object

–	**currenttime** num	returns a time value
–	**damagepath** –	sets path to damage path
–	**dumpsys** –	dump state to standard output
–	**emptypath** boolean	tests current path
–	**enumeratefontdicts** names	scans font dictionaries
–	**eoclipcanvas** –	eoclip to current canvas
dx dy	**eocopyarea** –	copy area to *dx, dy*
–	**eocurrentpath** shape	returns current path
canvas	**eoreshapecanvas** –	even/odd reshape of canvas
file *or* string	**eowritecanvas** –	write canvas to file
file *or* string	**eowritescreen** –	write screen to file
any	**errored** boolean	use like **stopped**
event	**expressinterest** –	queue input *events*
–	**extenddamage** –	extend damaged path
–	**eoextenddamage** –	extend damaged path
string1 string2	**file** file	same as Adobe implementation
proc	**fork** process	creates a new process
string	**forkunix** –	forks a UNIX process
canvas	**getcanvascursor** font char char	gets cursor for *canvas*
canvas	**getcanvaslocation** x y	returns canvas location
string1	**getenv** string2	gets value of *string1* in server
–	**geteventlogger** process	get event logger *process*
–	**getkeyboardtranslation** num	returns mode of translation
–	**getmousetranslation** boolean	are events translated?
file	**getsocketlocaladdress** string	return address of *file*
file	**getsocketpeername** string	return name of host connected
h s b	**hsbcolor** color	return color matching *h s b*
canvas	**imagecanvas** –	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas** –	analogous to **imagemask**
canvas x y	**insertcanvasabove** –	insert above current canvas
canvas x y	**insertcanvasbelow** –	insert below current canvas
–	**keyboardtype** num	return type of keyboard
process	**killprocess** –	kills *process*
process	**killprocessgroup** –	kills entire *process*group
–	**lasteventtime** num	returns **TimeStamp**
–	**localhostname** string	returns network hostname
a b	**max** c	leaves maximum on stack
a b	**min** c	leaves minimum on stack
monitor proc	**monitor** –	exec *proc* with locked *monitor*
monitor	**monitorlocked** boolean	checks state of monitor
x y	**movecanvas** –	moves canvas to *x y*
pcanvas	**newcanvas** ncanvas	creates a new canvas
–	**newprocessgroup** –	creates a new processgroup
array	**pathforallvec** –	analogous to **pathforall**
–	**pause** –	suspends current process
x y	**pointinpath** boolean	is *x y* in path?
string1 string2	**putenv** –	alter value of *string1*
–	**random** num	return random value
string	**readcanvas** canvas	read *string* as canvas

event	**recallevent** –	remove *event* from queue
event	**redistributeevent** –	enter *event* into queue
canvas	**reshapecanvas** –	sets *canvas* to be path
event	**revokeinterest** –	revoke interest in *event*
r g b	**rgbcolor** color	set color to *r g b* value
event	**sendevent** –	launch an *event*
boolean	**setautobind** –	set autobinding
canvas	**setcanvas** –	set current canvas
font char char	**setcanvascursor** –	set cursor identifiers
color	**setcolor** –	set current color
x y	**setcursorlocation** –	set cursor to *x y*
process	**seteventlogger** –	make *process* event logger
object integer	**setfileinputtoken** –	add *object* to tokenlist
boolean	**setkeyboardtranslation** –	is translation on?
n	**setlinequality** –	set linequality value
boolean	**setmousetranslation** –	sets mouse translation mode
path	**setpath** –	set path to *path*
boolean	**setprintermatch** –	set **printermatch** flag
num	**setrasteropcode** –	set rasterop combination function
graphicsstate	**setstate** –	set graphics state
–	**startkeyboardandmouse** –	initiate server processing
process	**suspendprocess** –	suspend *process*
num	**tagprint** –	put *num* on output stream
object	**typedprint** –	put *object* on output stream
–	**unblockinputqueue** –	release input queue block
dictionary key	**undef** –	undefine *key* from *dictionary*
process	**waitprocess** value	wait until *process* completion
file *or* string	**writecanvas** –	write canvas to *file*
file *or* string	**writescreen** –	write screen to *file*

C.2. NeWS Operators, by Type

The following operators are sorted by type.

Canvas Operators

w h bits matrix proc	**buildimage** canvas	constructs canvas object
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clipcanvas** –	clip to canvas boundary
–	**clipcanvaspath** –	set current path to clip
string	**createdevice** canvas	create new canvas
canvas	**createoverlay** canvas	create overlay canvas
–	**currentcanvas** canvas	current canvas
–	**eoclipcanvas** –	eoclip to current canvas
canvas	**eoreshapecanvas** –	even/odd reshape of canvas
file *or* string	**eowritecanvas** –	write canvas to file
file *or* string	**eowritescreen** –	write screen to file

canvas	**getcanvaslocation** x y	returns canvas location
canvas	**imagecanvas** –	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas** –	analogous to **imagemask**
canvas x y	**insertcanvasabove** –	insert above current canvas
canvas x y	**insertcanvasbelow** –	insert below current canvas
x y	**movecanvas** –	moves canvas to *x y*
pcanvas	**newcanvas** ncanvas	creates a new canvas
string	**readcanvas** canvas	read *string* as canvas
canvas	**reshapecanvas** –	sets *canvas* to be path
canvas	**setcanvas** –	set current canvas
file *or* string	**writecanvas** –	write canvas to *file*
file *or* string	**writescreen** –	write screen to *file*

Event Operators

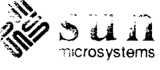
–	**awaitevent** event	blocks for *event*
num	**blockinputqueue** –	block input events
–	**countinputqueue** num	returns count of input queue
–	**createevent** event	create an event
event	**expressinterest** –	queue input *events*
–	**geteventlogger** process	get event logger *process*
–	**getmousetranslation** boolean	are events translated?
–	**lasteventtime** num	returns **TimeStamp**
event	**recallevent** –	remove *event* from queue
event	**redistributeevent** –	enter *event* into queue
event	**revokeinterest** –	revoke interest in *event*
event	**sendevent** –	launch an *event*
–	**unblockinputqueue** –	release input queue block

Mathematical Operators

num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
num	**arctan** num	computes arc tangent
a b	**max** c	leaves max on stack
a b	**min** c	leaves min on stack
–	**random** num	return random value

Process Operators

–	**breakpoint** –	suspends current process
process	**continueprocess** –	restart suspended process
–	**createmonitor** monitor	create monitor object
–	**currentprocess** process	return current process object
proc	**fork** process	creates a new process
string	**forkunix** –	forks a UNIX process
process	**killprocess** –	kills *process*
process	**killprocessgroup** –	kills entire *process*group
monitor proc	**monitor** –	exec *proc* with locked *monito*



monitor	**monitorlocked** boolean	checks state of monitor
–	**newprocessgroup** –	creates a new processgroup
–	**pause** –	suspends current process
process	**seteventlogger** –	make *process* event logger
	suspendprocess –	suspend *process*
process	**waitprocess** value	wait until *process* completion

Path Operators

dx dy	**copyarea** –	copy path to *dx, dy*
–	**currentpath** shape	return current path
–	**damagepath** –	sets path to damage path
–	**emptypath** boolean	tests current path
dx dy	**eocopyarea** –	copy area to *dx, dy*
–	**eocurrentpath** shape	returns current path
–	**extenddamage** –	extend damaged path
–	**eoextenddamage** –	extend damaged path
x y	**pointinpath** boolean	*x y* in path?
path	**setpath** –	set path to *path*

File Operators

listenfile	**acceptconnection** file	listens for connection
string1 string2	**file** file	same as Adobe implementation
file	**getsocketlocaladdress** string	return address of *file*
file	**getsocketpeername** string	return name of host connected
num	**tagprint** –	put *num* on output stream
object	**typedprint** –	put *object* on output stream

Color Operators

color	**contrastswithcurrent** boolean	compare colors
–	**currentcolor** color	current color
h s b	**hsbcolor** color	return color matching *h s b*
r g b	**rgbcolor** color	set color to *r g b* value
color	**setcolor** –	set current color

Keyboard and Mouse Operators

–	**currentcursorlocation** x y	returns mouse coordinates
–	**getkeyboardtranslation** num	returns mode of translation
–	**getmousetranslation** boolean	are events translated?
–	**keyboardtype** num	return type of keyboard
boolean	**setkeyboardtranslation** –	is translation on?
boolean	**setmousetranslation** –	sets mouse translation mode
–	**startkeyboardandmouse** –	initiate server processing

Cursor Operators

–	**currentcursorlocation** x y	returns mouse coordinates
canvas	**getcanvascursor** font char char	gets cursor for *canvas*
font char char	**setcanvascursor** –	set cursor identifiers
x y	**setcursorlocation** –	set cursor to *x y*

Miscellaneous Operators

–	**currentautobind** boolean	autobinding enabled?
–	**currentlinequality** n	current line quality
–	**currentprintermatch** boolean	return **printermatch** value
–	**currentrasteropcode** num	rasterop combination function
–	**currentstate** state	returns **graphicsstate** object
–	**currenttime** num	returns a time value
–	**dumpsys** –	dump state to standard output
–	**enumeratefontdicts** names	scans font dictionaries
any	**errored** boolean	use like **stopped**
string1	**getenv** string2	gets value of *string1* in server
–	**localhostname** string	returns network hostname
array	**pathforallvec** –	analogous to **pathforall**
string1 string2	**putenv** –	alter value of *string1*
boolean	**setautobind** –	set autobinding
n	**setlinequality** –	set linequality value
boolean	**setprintermatch** –	set **printermatch** flag
num	**setrasteropcode** –	set rasterop combination function
graphicsstate	**setstate** –	set graphics state
dictionary key	**undef** –	undefine *key* from *dictionary*

D

NeWS Manual Pages

NeWS Manual Pages **213**

bldfamily(1) — build font family description 213
cps(1) — construct C to POSTSCRIPT language interface 215
dumpfont(1) — dump font in another format 217
journalling(1) — record and playback package 219
kbd_mode(1) — change keyboard translation mode 221
news_server(1) — NeWS server program 223
psterm(1) 225
newsdemos(6) — NeWS demonstrations 231
newshost(1) — NeWS network security control 237
psh(1) — NeWS POSTSCRIPT shell 239
psload(1) — display load average under NeWS 241
man — on-line display of reference pages using NeWS 243
psterm(1) — NeWS terminal emulator 245
psview(1) — POSTSCRIPT language previewer 247
say(1) — execute POSTSCRIPT language fragment 249
setnewshost(1) — set NEWSSERVER environment 251
xdemos(6) — X Window System demonstration 253
psio(3) — NeWS buffered input/output package 255

NAME

bldfamily – build font family description

SYNOPSIS

bldfamily [–**d***dirname*] [–**o***outname*] [–**v**] *names*

DESCRIPTION

bldfamily scans a sets of NeWS font files and produces a NeWS font family file. A font family is a set of font files that are grouped together to provide a single POSTSCRIPT language font. In the POSTSCRIPT language, a font has a name, like Times-Roman, and can be rendered in many different sizes. A NeWS font file is an instance of a POSTSCRIPT language font at a particular size. Font family files contain the information necessary for NeWS to pick the right bitmap font.

bldfamily scans directory *dirname* for files with extension ".fb" or ".fm" and where the leading non-digit characters of the filename match one of the *names* . The family file that is built will be written to *dirname*/*outname* **.ff.**

If *outname* isn't specified, it defaults to the first of the *names* . If *dirname* isn't specified, it defaults to "$FONDIR" if defined, "." otherwise. If *names* isn't specified *bldfamily* scans *dirname* for all font files and builds all the possible family files.

OPTIONS

–**d***dirname* Specifies the directory to scan and put the **.ffam** file into.

–**o***outname* Specifies the output font name

–**v** Verbose – gives a more detailed description of what is going on.

EXAMPLE

paper% **dumpfont -d /usr/newfonts -n Boston boston*.vfont**
paper% **bldfamily -d/usr/newfonts Boston**

The first command calls *dumpfont* and converts all the vfont files whose names match "boston*.vfont" into NeWS format. It puts them into */usr/newfonts* and changes their font name to Boston (it would have defaulted to boston). The second command calls *bldfamily* and scans */usr/newfonts* for "Boston*.fb" and builds a font family file for them, which will be called */usr/newfonts/Boston.ff.*

SEE ALSO

dumpfont(1)

DIAGNOSTICS

filename isn't a font file

bldfamily has found a file that matches one of the names and has an appropriate extension, but it isn't a valid font file. You should probably ignore this message.

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

cps – construct C to POSTSCRIPT language interface

SYNOPSIS

cps [**–c**] [**–D** *symbol*] [**–I** *filename*] [**–i**] [*POSTSCRIPT language file*]

DESCRIPTION

CPS compiles a specification file containing procedure names and POSTSCRIPT language code into a header file (`filename.h`) that can be included in programs. Only one input file can be specified, and if the `filename.h` file has been previously created, a backup copy of this file will be generated in the form `filename.h.BAK` before the new file is generated.

The convention is for the input specification file to end in `.cps`.

OPTIONS

–c
Compiles a POSTSCRIPT language file for faster loading by NeWS, and is not used to generate a specification file for programs. For example, the following command line:

```
cps -c < input_file > output_file
```

will convert the *input_file* from the ascii form of the POSTSCRIPT language, to the compressed binary form. When read by NeWS, the *output_file* will execute exactly the same as *input_file*, except that it will be faster. The **–c** option **will not** work if the *input_file* uses constructs like currentfile readstring, which are often used with the image primitive.

–D *symbol*
Defines symbols to be passed onto the language pre-processor (`cpp`) which processes the input file.

–I *filename*
Specifies include files. Passed on to the pre-processor.

–i
Generates two specification files: one that contains only the procedures and POSTSCRIPT language code that are user-defined, and one that contains other definitions required for the C-POSTSCRIPT language interface. For example, and would be defined in the second file. The second file references the user-defined procedures as `extern char`. The first file is of the form *filename.c*, and the second file is of the form *filename.h*. `.BAK` files will be generated if the files already exist.

This option is valuable for controlling the size of the CPS include files in multiple source files. The *filename.h* would only need to be included once. Each source file would only need to include it's specific *filename.c* file generated by this option.

SEE ALSO

NeWS Manual - Chapter 9 "C Client Interface"
cpp(1)

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems, Inc.

NAME

dumpfont – dump font out in some other format

SYNOPSIS

dumpfont [**–a**|**–b**|**–v**|] [**–c** *comment*] [**–d** *dirname*] [**–f** *n*] [**–n** *fontname*] [**–S**] [**–s** *n*] [**–t**] [**–tv**] [**–ta**] *filenames*

DESCRIPTION

dumpfont reads in the set of named font files and dumps them out again according to the specified options, effectively converting the files from one font format to another. *dumpfont* is typically used to generate fonts for use with the NeWS window system.

There are five types of font files that *dumpfont* can read: Sun standard vfont format, Adobe ASCII bitmap format, Adobe ASCII metric format, NeWS font format, and CMU (Andrew) format. The format of the input font is determined automatically by inspecting the file. It can write fonts out in one of three formats: Adobe ASCII, NeWS, and vfont. The default output format is NeWS.

OPTIONS

–a Selects Adobe ASCII output format. This is the format that you should use when transporting fonts from one machine architecture to another. The output file extension will be ".afont".

–b Selects NeWS output format (the default). The output file extension will be ".font". If the input file is an Adobe ASCII metrics file, the extension will be ".metrics".

–v Selects vfont output format. The output file extension will be ".vfont".

–vf Selects vfont output format. The output file extension will be ".vfont". Forces the characters to be fixed width.

–c *comment* Sets the *comment* field of the font. The Adobe ASCII and NeWS font formats support an internal comment that accompanies the font. This is usually used to contain copyright or history information. It is normally propagated automatically.

–d *dirname* Specifies the directory into which the font files will be written. If the FONTDIR environment variable is set, it is used as the default value. Otherwise, if the NEWSHOME environment variable is set, *$NEWSHOME/fonts* is used as the default value. Otherwise "." is used.

–f *n* Sets the maximum length of an output filename (excluding extension) to *n* . When writing NeWS format files, NeWS normally constructs the output filename from the name of the font and its scaling factors. Some systems cannot cope with long file names, so this option can be used to heuristically squeeze the name.

–n *name* Forces the output font name to be *name* . It is important to not confuse the name of the font with the name of the file that contains it. Some font formats (Adobe ASCII and NeWS) contain the name of the font internally. So given a 10-point Times-Roman font, its font name will be "Times-Roman", but its file name might be *TimRom10.font.*

–S Attempts to determine the size information of fonts by inspecting the bitmaps and applying some heuristics. This is useful when reading vfonts (particularly those intended for printers like the Versatec) that are missing or have incorrect size information.

–s *n*	Sets the point size of the font to *n* . Overrides any internal size specification
–t	Prints a short description of the fonts on standard output; a reformatted font file is no dumped.
–tv	Prints a move verbose description of the fonts on standard output; a reformatted font file is not dumped.
–ta	Prints a long description of the fonts on standard output; a reformatted font file is no dumped. You'll get every scrap of information.

SEE ALSO

bldfamily(1), vfont(5)

DIAGNOSTICS

Bad flag: -C	Unknown command like option
Couldn't write ...	Error writing font file
f: not a valid font.	Unknown input file format.

BUGS

Should have been named *convertfont* .

NAME

journalling – NeWS event record and playback package

SYNOPSIS

journalling

DESCRIPTION

The Journalling package allows you to capture NeWS mouse and keyboard events onto a file, and then play the file back. This results in NeWS acting like a player piano, faithfully duplicating the original user actions in real time.

This package permits continuous replaying of a given file. Playback can be interrupted at any time by clicking one of the mouse buttons.

Journalling also includes playback speed control, which allows you to slow down or speed up the playback rate.

USAGE

Invoking the *journalling* program will add a *Journalling =>* menu item to the root menu. The *Journal* menu item in the NeWS *Applications =>* sub-menu does the same thing. Note that it will take a few seconds for journalling to load everything into the NeWS server.

There are five submenus under the main Journalling menu item:

Control Panel - Brings up the Journalling control panel

Start Recording - Start recording on the current Record file

Stop Recording - Signals the end of recording

Playback - Starts playback of the current Playback file

Remove Journalling - Gets rid of all journalling menus and resources

Note that Playback can be interrupted at any time by hitting one of the mouse keys.

Selecting the Control Panel item brings up a control panel window. It contains the following items:

RECORD, STOP, and PLAY buttons: These buttons perform the same function as the corresponding menu items. They also light up to indicate what action is currently taking place. They can be used interchangeably with the menu items.

Record File: This text item allows you to specify the current file to record onto. It can be any valid filename on the server machine. Relative pathnames are taken to be relative to the directory that NeWS was started from. The default for the Record file is */tmp/NeWS.journal*.

Playback File: The current file to playback from. It has the same characteristics as the Record File.

Play Forever toggle switch: If this switch is on then Journalling will automatically repeat playing the Playback File.

Playback Speed: Slider that scales the playback time. Positive values make playback speed up, negative values make playback slow down. This facility is dependent on the speed of the underlying hardware. It is not calibrated between different machines.

Done: The Done button will hide the Control Panel. It can be brought back up by selec
Control Panel menu item. The *Zap* window command has the same effect.

TIPS FOR USING JOURNALLING

When creating journals that will be replayed repeatedly, it is important to get rid of whatever windows y have created at the end of the journal. The state of the screen should be just as it was when the journal v begun. Otherwise, the NeWS server will eventually run out of memory because you are continually cre ing new windows. Doing a *Zap* from the *All Windows* menu at the end of the journal will provide desired effect. However, be sure that you then restart a Console window to catch system messages.

There is a noticeable variation in performance on NeWS running on different kinds of machines; it r much faster on a Sun 4 than on a 3/50! This means that playing back a script recorded on a fast mach might not always work correctly on a slower machine. A given machine can handle NeWS events at sc maximum rate. The Playback Speed Control will allow you to adapt playback speed of a given script t fairly wide range of machines; unfortunately, this requires a bit of trial and error.

Care must be taken when recording sequences that contain invocations of Unix programs, particul when starting new applications. The mouse must not be clicked until the bounding box is up on the scre If the mouse is clicked early, the wrong window sizing will be made on playback, leading to unpredict behavior due to the window not being where it was when recording.

FILES

${NEWSHOME}/lib/NeWS/journal.ps — Contains the low level journalling code, the control pa and menu code, and state button Lite Item code used the control panel.

${NEWSHOME}/demo/journalling — Loads *${NEWSHOME}/lib/NeWS/journal.ps*.

SEE ALSO

NeWS Manual

BUGS

The unpredictable behavior of playback due to the non-deterministic Unix scheduling mechanism and g eral operating environment make reliance on the Journalling package for critical functions unadvisable.

NAME

kbd_mode – change the keyboard translation mode

SYNOPSIS

kbd_mode −a | −n | −e | −u

DESCRIPTION

kbd_mode sets the translation mode of the console's keyboard (/dev/kbd) to one of the four values defined for KIOCTRANS in *kb*(4S). This is useful when a program which resets the translation mode crashes; for example, NeWS (when run from SunView) can sometimes leave SunView reading untranslated events.

OPTIONS

−a ASCII: the keyboard will generate simple ASCII characters

−n none: the keyboard will generate unencoded bytes – a distinct value for up and down on each switch on the keyboard

−e events: the keyboard will generate SunWindows input events with ASCII characters in the *value* field

−u unencoded: the keyboard will generate SunWindows input events with unencoded bytes in the *value* field (this is the mode NeWS currently uses).

FILES

/dev/kbd
$NEWSHOME/bin/kbd_mode

SEE ALSO

kb(4S)

NAME

news_server — NeWS server

SYNOPSIS

news_server [POSTSCRIPT]

DESCRIPTION

The *news_server* command starts the NeWS server. The NeWS server is an interpreter for a subset of the POSTSCRIPT language. The POSTSCRIPT language was defined by Adobe Systems Inc. NeWS supports many overlapping drawing surfaces, multiple lightweight threads of execution, and message-based interprocess communication. Details of the the structure of NeWS are found in the *NeWS Manual*.

OPTIONS

[POSTSCRIPT]

The NeWS server interprets the POSTSCRIPT program given as an argument on the command line. If no POSTSCRIPT language text is given on the command line, *news_server* executes

(NeWS/init.ps) (r) file cvx exec &main

This POSTSCRIPT language fragment sets up NeWS for its normal use as a window server. When specifying this argument, you probably should put single quotes around the "s fragment " to protect it from premature interpretation by the shell. The POSTSCRIPT language files **%stdin**, **%stdout**, and **%stderr** have their normal meanings while this program is executing.

FRAMEBUFFER

Another option is defined by the **FRAMEBUFFER** environment variable. The device name (e.g., */dev/bwtwo0, /dev/cgtwo0*) contained in **FRAMEBUFFER** tells NeWS which frame buffer to display on. If **FRAMEBUFFER** is not defined then */dev/fb* is the default.

USAGE

The first thing that you should do is to start a terminal emulator which acts as a console. This console window will display system messages and will prevent these messages from writing over your display. Bring up a console window by depressing the right mouse button and sliding the mouse to the right until the word 'Console' is displayed under the mouse cursor. Now release the right mouse.

Compatibility

NeWS may be run either from outside the SunView environment, or from inside the SunView environment by using *overview*(1). **NB**: You must be sure that the **FRAMEBUFFER** environment variable used by the NeWS server matches the **-d** argument used by server matches the **-d** argument used by `suntools`. Both of these values default to `/dev/fb`, so if one is changed, the other must be also:

overview -w news_server

While running NeWS from outside SunView, SunView tool binaries may be executed as normal. Their windows will appear on top of all NeWS windows, but will otherwise behave normally.

Root Menu

When NeWS starts, using the standard *init.ps*, it paints the desktop gray and waits for the user to select a menu option. By default, menus pop up on the right mouse button. The standard root menu is (shown with the submenus hierarchy expanded):

```
Applications =>
        Terminals =>
                Fixed Size =>
                        Console
                        sun
                        H19
                        bitgraph
```

            - vt100
            - wyse
            - tvi925
        - Console
        - sun
        - H19
        - bitgraph
        - vt100
        - wyse
        - tvi925
    - Clocks =>
        - Plain
        - Plain (seconds)
        - Fancy
        - Fancy (seconds)
    - Load Average
    - Calculator
    - Journal
- Demos =>
    - (see newsdemo(6))
- All Windows =>
    - Zap
    - Open
    - Close
    - Flip
    - Tidy
    - Here
    - Drop
    - Top
    - Bounce =>
        - Windows
        - Icons
        - All
        - Stop
- SunView1 =>
    - Selection Transfer =>
        - NeWS to SunView Shelf
        - SunView to NeWS Shelf
    - Applications =>
        - shelltool
        - cmdtool
        - mailtool
        - textedit
        - defaultsedit
        - iconedit
        - dbxtool
        - perfmeter
        - clock
        - gfxtool
        - console
        - lockscreen
        - Default .suntools
- User Interface =>

- Input Focus =>
  - Click to Type
  - Follow Cursor
- Window Management Style =>
  - Rubber-band Box =>
    - Thin
    - Thick
    - Grid
  - Zoom =>
    - Zoom Slow
    - Zoom Medium
    - Zoom Fast
  - Flip Drag
- Look & Feel =>
  - NeWS Default
  - SunView1
- Retained Windows =>
  - On
  - Off
  - Default
- Root Image =>
  - Group
  - Plain

Repair =>
- Repaint All
- Reset Input

Exit NeWS =>
- No, not really.
- Yes, really!

Applications =>

This menu lets you start up a number of NeWS applications.

Terminals is a pull-right menu from which you can create a window using the *psterm*(1) terminal emulator program. If you select one of these terminals, you must drag out the exact size of the window in which the terminal emulator will run. There is another menu item called *Fixed Startup*, which has the same options as the *Terminals* menu. If you select one of the terminals from *Fixed Startup*, you need not specify the size and location of the terminal emulator; it will be created automatically in the lower-left corner of the screen. Selecting *Console* from either the *Terminals* menu or the *Fixed Startup* menu creates an H19 terminal emulator that is set up to receive console messages.

Clocks is a pull-right menu from which you can create a clock. The *Plain* clock is a simple round clock. The *Fancy* clock is a stylish modern round clock. Both clocks can be create with an option to show the seconds.

Load Average invokes the *psload* program. This program is a load average monitor.

Journal causes a journalling mechanism to be loaded into NeWS. In addition, a menu entitled ''Journalling'' is installed in the top level root menu. The journalling mechanism allows you to capture and playback user actions. See *journalling(1)*.

Demos =>

Lets you run one of the many demonstration programs. They are described in *newsdemos*(

All Windows =>

Provides you with the choices necessary to manage all the windows on your display at or Some of the operations are more fun than useful.

Zap causes all the windows on the screen to be destroyed.

Open causes all the windows on the screen to be opened.

Close causes all the windows on the screen to be closed.

Flip causes all the windows on the screen to toggle between their open state and their closed st

Tidy causes all the open windows on the screen to be moved to their closest corner. All clo windows line up along the bottom edge of the screen.

Close causes all the windows on the screen to be closed.

Here Allows you to position all the windows on the screen with successive clicks of the mouse

Drop causes all the windows on the screen to ''fall'' to the bottom of the screen.

Top causes all the windows on the screen to ''rise'' to the top of the screen.

Bounce is a pull-right menu that is used to invoke a demo that is used to ''bounce'' wind around the screen. This demo is particularly slick when all the windows are retained. bounce just the open windows with the *Windows* command. You can bounce just the icons the *Icons* command. You can bounce both with the *All* command. *Stop* terminates the bouncir

SunView1 =>

As mentioned above, you can run existing SunView1 and SunWindows application concurre with NeWS applications.

Selection Transfer lets you exchange selections between NeWS applications and SunView appl tions. *NeWS to SunView Shelf* takes the current NeWS selection and loads it onto the Sun shelf. A subsequent *get* operation in SunView will retreive the contents of the shelf. *SunVie NeWS Shelf* takes the current SunView selection and loads it onto the NeWS shelf. A subseq *get* operation in NeWS will retreive the contents of the shelf. **Note**: There is a race cond between when you invoke one of these shelf transfer menu operations and when you subseque release the *get* key. Waiting a second or two between the two operations should avoid any p lem.

Applications lets you invoke any of the following SunView applications: *shelltool, cmdtool, tedit, mailtool, defaultsedit, iconedit, dbxtool, perfmeter, clock, gfxtool, console, lockscreen. Default .suntools* entry starts up the standard set of SunView applications, which includes a sole window, a text editor, a clock, a mail handler, and a terminal emulator.

User Interface =>

The *User Interface* menu lets you alter a number of user interface options.

Input Focus is a pull-right that provides you with a choice of keyboard focus mechanism *to Type* forces input to occur in whichever window you last clicked the mouse (independe

current cursor location). *Follow Cursor* forces the input focus to shift to whichever window the cursor is currently in.

Window Management Style pull-right provides you with a number of options for controlling the default behavior of your windows. The *Rubber-band Box* menu controls the appearance of the rubber-band rectangle that appears when you specify window size; the choices are *Thin*, *Thick* and *Grid*. The *Zoom* menu controls the speed with which windows zoom into icons and vice versa; the choices are *Zoom Slow*, *Zoom Medium* and *Zoom Fast*. *Flip Drag* toggles the method of moving windows between dragging the entire window and dragging just its frame.

Look & Feel is a pull-right that allows you to toggle between the standard NeWS look and feel and the SunView1 look and feel. Both windows and menus are altered to reflect the chosen look and feel.

Retained Windows is a pull-right that allows you to choose the image saving behavior of newly created windows. If you turn retention *On*, windows redisplay very quickly, at the expense of more main memory usage. If you turn retention *Off*, applications redisplay by repainting their image, not by letting the system do it from an image stored off-screen. The *Default* setting is retained for one bit deep frame buffers but not for deeper frame buffers.

Root Image is a pull-right that controls the color or pattern on the background (root) window. Select *Plain* to get your root window back to the default background. Select *Group* to show an image that mentions NeWS and shows the Sun logo.

Repair =>
: Allows you to reset the input mechanism (*Reset Input*) or to repaint all the windows (*Repaint All*).

Exit NeWS =>
: This menu allows you to exit NeWS gracefully. It is done as a menu in order to avoid accidental invocation.

Window Management

The standard window management package provides for window manipulation via a pop-up menu invoked from within the frame of a window. Certain functions are also available via accelerators. Here is the window management menu (it is found under the *Frame* pull-right):

Move
: Lets you move the window by dragging it around with the mouse. Click a mouse button to leave the window at its new location.

Move Constrained
: Lets you move the window, with motion constrained either vertically or horizontally. If you click in the left or right part of the window's frame, you can move the window only horizontally. If you click in the top or bottom part of the window's frame, you can move the window only vertically. Click once more to leave the window at its new location.

Top
: Raises this window above all of the other windows.

Bottom
: Pushes this window below all of the other windows.

Zap
: Causes the current program to exit.

Resize
: Allows you to change the size and placement of a window. Click where you want the upper left corner of the window to be, then drag out a rubber-band rectangle and click where you want the lower right corner to be, just as if you were creating a new window. The window will be resized and moved to the rectangle you just swept out.

Stretch Corner
: Allows you to change a window's size by dragging one corner, leaving the opposite corner fixed where it is. Click a button when the mouse is near the corner you wish to drag; then click again

when the rubber-band rectangle is the size you want it to be.

Stretch Edge
: Allows you to drag a window's edge, just like dragging a corner. Click near the edge you want t drag, and then click again when you have placed it.

Close
: Closes this window into an icon.

Redisplay
: Causes the window to redisplay itself.

You can use accelerators instead of the menu to manage windows. To raise a window, click the left butto in the window's frame. To drag a window, press the middle button in the window's frame, and drag th window into position with the button still down. To close the window into an icon, click the left button i the cycle glyph in the upper left corner of the window. To resize a window, click the left button in th resize glyph in the lower right corner of the window.

You can use similar management functions on icons as well as on windows. You can bring up a windo management menu over an icon by holding down the right button anywhere in the icon. This menu is sim lar to the menu for windows, except that it contains the following entries: *Move*, *Top*, *Bottom*, *Zap*, *Ope Open&Resize*, and *Redisplay*. The only new function is *Open&Resize*. This function opens an icon into window that you drag out with the mouse. This is in contrast to *Open*, which causes the window to ope into its original shape. You can use accelerators on icons, too. You can click the left mouse button on a icon to open it, and you can drag an icon to a new location with the middle button.

Interpreter Access

The standard **init.ps** file starts a server listening on port 144 (and then 2000) for client connections. Yo can use a program called *psh*(1) to connect to this port. At this point, you are talking to a POSTSCRI language interpreter, and you can interact with it in the normal way. However, any errors you mak u the server to break the connection. To put yourself into an environment that is more forgiving of error you must type **executive** after you connect. For example,

```
paper% psh
executive
Welcome to NeWS Version 1.1
```

Type **quit** to the interpreter to exit *psh*.

Remote Access

See *newshost(1)* for information about allowing remote hosts to start applications on your local machin The default is that other machines can't open connections to your NeWS server.

NeWS Sockets

If there is no socket specified in the `NEWSSOCKET` environment variable or in `user.ps`, NeWS will t to listen on socket `144`. Since `144` is a privileged socket, unless `news_server` is running as root, t attempt to listen on `144` will fail, and the NeWS server will then try to listen on socket `2000`. This ord of consideration may be overridden by setting the NEWSSOCKET environment variable. The followir shell archive allows you to run two NeWS servers on two displays:

```
#! /bin/sh
FRAMEBUFFER=/dev/bwtwo0 NEWSSOCKET=%socketl2000 news_server &
FRAMEBUFFER=/dev/cgtwo0 NEWSSOCKET=%socketl2001 news_server &
sleep 5
adjacentscreens /dev/bwtwo0 -r /dev/cgtwo0
```

The socket on which NeWS listens can also be set in your `user.ps` file with a line of the form:

/NeWS_socket (%socketl2001) def

This will override any socket specified in the NEWSSOCKET environment variable.

FILES

If *news_server* is unable to open a file whose name doesn't start with /, and which can't be found in your home directory or the directory you started NeWS from, it inserts *${NEWSHOME}/lib* at the front of the name and tries again.

${NEWSHOME}/lib/NeWS/*.ps	Startup POSTSCRIPT language programs.
${NEWSHOME}/fonts/*	Font Library
${NEWSHOME}/bin/news_server	the NeWS server
~/user.ps	user-definable server customizations; loaded after other system *.ps files
~/startup.ps	user-definable server customizations; loaded before other system *.ps files

The **FRAMEBUFFER** environment variable specifies the default frame buffer for NeWS. It defaults to */dev/fb*.

SEE ALSO

psh(1), psterm(1), psview(1), say(1), newsdemos(6), xdemos(6), journalling(1), kbd_mode(1), newshost(1), psload(1), psman(1), setnewshost(1)

NeWS Manual

PostScript Language Reference Manual, Adobe Systems Inc., Addison-Wesley

BUGS

Some parts of the POSTSCRIPT language have yet to be implemented. See the chapter in the *NeWS Manual* entitled *Omissions and Implementation Limits.*

The NeWS server is not yet completely robust when it runs out of memory. This out of memory condition occurs because swap space has been used up. Swap space is a resource that is shared by all the processes running on your machine. NeWS has been designed to print a message on the console about being low on memory when it gets very close to being out of memory. If you see this message then you should immediately reduce the stress on swap space by terminating some large processes [see pstat(8) with the -s flag]. If NeWS does completely run out of memory, and it can't recover, then it is designed to print a message on the console about being out of memory and aborting.

When running SunView1 program, you may see many ''Window display lock broken...'' messages. You should have a console window for these message to appear in so as to avoid trashing the screen.

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

NeWS demos – NeWS demonstrations

SYNOPSIS

Demos menu item in the NeWS root menu

OVERVIEW

The **Demos** pull-right menu provides access to several NeWS demonstration programs. These programs are intended to demonstrate NeWS graphics and user interaction capabilities.

Many of these programs are written in the extended version of the POSTSCRIPT language understood by NeWS, using the *psh*(1) program. *psh* simply opens a connection to the NeWS server and sends NeWS commands to it.

Unless described otherwise, you must specify a window for the demo when you start it. When a small box appears by the cursor, you should *click* (and let go of) the right mouse button to indicate where you want one corner of the window to be. Then, as you move the mouse, a *rubber-band* box expands and contracts to show a window region. Click the right mouse button again to indicate where you want the opposite corner to be. You should not hold down the right button while defining the box.

DESCRIPTION

The **Demos** menu is available as a pull-right item from the NeWS root menu. This menu contains the following entries:

- Animation =>
  - Bounce
  - Spin
  - Wink
  - Icosahedron
  - Icosahedron screensaver
- Color =>
  - Color Cube
  - Color Wheel
  - Color Names
- Games =>
  - Go
  - Backgammon
- Images =>
  - Display Scanned Image
  - Image Rotate
  - Image Scale
  - Image Stencil
  - Image Spin
  - Catalyst
- Line Drawing =>
  - Escher's fish
  - Lines
  - Rubber-band
  - Vectors
  - World
- Miscellaneous =>
  - Flags
  - Item Demo
  - Pie Chart
  - Spiral
- Previewer =>
  - Golfer

- Rose
- Shuttle
- Nozzle
- Overview

Text =>

- Text
- Text (scaled)
- Language Demo
- Icon Browse

X.10 Demos =>

- Run a Demo =>
  - Analog Clock
  - Load Average
- Kill X Server

Animation =>

The **Animation** pull-right item brings up a menu that contains several demonstrations of NeW animation capabilities.

Bounce bounces a moving puck around inside of a window. You can use the menu inside the w dow to stop and start the puck, to change its size and color, and to change the speed at which puck bounces. You can drag the *Bounce* window while it is running; this demonstrates NeW lightweight process mechanism.

Spin displays a spinning globe. It is best to select a small (1 inch x 1 inch) area for this de

Wink displays a pair of eyes in the middle of the screen, one of which winks at you.

Icosahedron displays a bouncing 20 sided regular solid with the hidden lines removed. Due to computation necessary to figure the hidden lines, this demo may run faster if the program is run another machine.

Icosahedron screensaver is like *icosahedron*, but runs on top of all visible window. This progr goes away with a click of a mouse button.

Color =>

The **Color** pull-right item brings up a menu that contains programs that demonstrate some NeWS' color capabilities. Other programs will display in color, what distinguished the progra in this menu is their focus on color.

Color Wheel draws a wheel of colors inside a window. If NeWS is running on a monochro display, it uses gray values instead of colors. You can use the menu to switch between gray color, and to vary the number of shades, the saturation, and the intensity of the colors displayed

Color Cube is similar to the *Color Wheel* in that it displays colors and gray levels; howeve presents them in a different format. Its menu lets you alter the presentation of the colors i manner similar to *Color Wheel*.

Color Names shows you the correspondence between color names in the color dictionary implemented by *NeWS/colors.ps*) and their colors on the screen. This program uses scrollbar access all the colors. In addition, there is an interesting use of the lite menu package to horizontal menu bar and menus with rows and columns. The menu is used to control text co font, face and size.

Games =>

The **Games** pull-right item brings up a menu with available games.

Go is a simple program that puts up a *Go* board with which you can interact by placing and removing stones. Thus, you could play a game of *go* with someone else while seated in front of the screen. It is intended as a simple (but complete) application to show programmers how C, CPS, and NeWS interact. The tutorial part of the *NeWS Tutorial and Cookbook* describes the internals of this program.

Backgammon is a game that puts up a *backgammon* board and will actually play against you. This program is a port of *gammontool*, so see its man page for further documentation.

Images =>

The **Images** pull-right item brings up a menu that contains several demonstrations of NeWS' imaging capabilities. Each program (except for *Image Scale*) lets you select the image to display by means of a pop-up menu. You bring up the menu by clicking the right mouse button inside the window (not in the window's frame).

Display Scanned Image creates a window and displays an image inside of it. The image will be scaled to fit exactly within the window boundaries, regardless of its original aspect ratio.

Image Rotate displays ten rotations of an image in a pinwheel arrangement.

Image Scale takes the bitmap image of a turkey (from the *PostScript Reference Manual*, page 171) and scales it as many times as will fit inside the window.

Image Stencil demonstrates NeWS' capability of pushing an image through an arbitrary path, or *stencil*. The right button menu brings up a menu giving a choice of several stencils in addition to the selection of the image to be displayed.

Image Spin demonstrates NeWS' image rotation capability. After you bring up the window, the server waits for you to define another rubber-band square with the mouse. Press the right mouse button where you want the lower left corner of the image to go and then release the button where you want the lower right corner of the image. The image is rotated and scaled to fit between the two points. The right button menu has an additional menu item, *Spin*, which lets you specify a different rotation for the image.

Catalyst shows images digitized from Sun's *Catalyst* Third Party Software catalog.

Line Drawing =>

This pull-right item brings up a menu with demonstrations of line drawing.

Escher's fish draws the famous *Square Limit* created by M. C. Escher. The demo is a 260-line recursive NeWS program that draws a large number of vectors. You can use the menu to vary the complexity of this drawing.

Lines creates a window with a line pattern inside of it. You can can alter the number of lines drawn from the pop-up menu inside the window. On color screens, the line pattern is displayed in a rainbow of colors.

Rubber-band demonstrates how responsive NeWS can be when interacting with you. When you bring up the window, NeWS draws a rubber-band line from a corner of the window to the current mouse location. This line will track the mouse as you move it around on the screen. When you

click a mouse button, NeWS tracks the mouse with a rubber-band curve instead of a li C more click kills the window.

Vectors is a demonstration of NeWS' vector-drawing capabilities. The demo draws four spa ships inside its window, composed of over 7,000 vectors.

World displays a geographic projection of the Western Hemisphere.

Miscellaneous =>

This pull-right item brings up a menu with miscellaneous demonstrations.

Flags displays flags of many nations, in color if possible. You can use the menu to display ju: single flag or all the flags at once.

Item Demo is a demonstration of user interface items. The set of items includes buttons, slid cycles, and text areas. All of the items can be dragged around with the mouse.

Pie Chart draws a business pie chart with slices of the pie filled with varying colors.

Spiral draws a simple spiral pattern.

Previewer =>

This pull-right item brings up a menu with demonstrations of NeWS' POSTSCRIPT language] viewing capabilities. The program *psview* is used to display POSTSCRIPT language files ou from other programs, e.g., *Frame's Frame Maker*, *AutoCAD*, and *Adobe's Illustrator*.

Golfer and *Rose* were produced using *Adobe's Illustrator* program.

Nozzle and *Shuttle* were produced using *AutoCAD*.

Overview was produced using *Frame's Frame Maker* program. This is a multi-page docun that provides an overview of NeWS' capabilities.

Text =>

This pull-right item brings up a menu with demonstrations of NeWS' text capabilities.

Text writes text inside a window in several styles. The right button brings up a pop-up menu 1 which you can select the font (under the *Font* pull-right), the point size, the colors, and the te be shown. The text shown can be either some sample text or a list of all characters in the ch font.

Text (scaled) demonstrates NeWS' ability to simulate the arbitrary scaling of text using onl map fonts. The line spacing and intercharacter spacing are varied so that a continuous ran sizes can be simulated with a fixed number of bitmap fonts.

Language Demo shows that NeWS can support several different languages. You can select o a variety of languages from the pop-up menu, causing both the menu and the text to be displ in the chosen language.

IconBrowse brings up a large window that displays icons from the Icon font. You use th control the range of characters displayed and to change the font from which they are disp

X.10 Demos =>

This pull-right item brings up a menu with a demonstration of a partial X.10 emmulation package. See *xdemos.6* for further information.

FILES

$NEWSHOME/lib/NeWS/demomenu.ps
NeWS code for the demo menu and some of the demo programs.

*$NEWSHOME/demo/** demo programs not built into the demo menu.

SEE ALSO

psh(1), psterm(1), psview(1), say(1), xdemos(6)

NeWS Manual

PostScript Language Reference Manual , Adobe Systems Inc., Addison-Wesley

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

newshost – NeWS network security control.

SYNOPSIS

newshost add [*hosts*]
or **newshost remove** [*hosts*]
or **newshost show**

DESCRIPTION

Newshost is a shell command that manipulate the registry of hosts that are allowed to connect to the NeWS server. The identity of the NeWS server whose registry will be manipulated is determined by the NEWS-SERVER environment variable. The variable **/NetSecurityWanted** (in the NeWS **systemdict**) may be set to false to disable the security mechanism.

newshost add	adds the named hosts to the registry,
newshost remove	removes the named hosts from the registery,
newshost show	prints out a list of the hosts in the registry.

SEE ALSO

NeWS Manual

NAME

psh – NeWS POSTSCRIPT shell

SYNOPSIS

psh [**files**]

DESCRIPTION

psh opens a connection to the NeWS server and transmits the file arguments (or stdin if no files are specified) to it. Any output from NeWS is copied to stdout. The files should be POSTSCRIPT programs for the NeWS server to execute.

A common use for *psh* is in creating applications written entirely in the POSTSCRIPT language. First, type your POSTSCRIPT program into a file. Then, type as its first line:

```
#! /usr/NeWS/bin/psh
```

If you now make the file executable (with *chmod)* you can invoke it by name from the shell, and UNIX will use */usr/NeWS/bin/psh* to execute it. *psh* will in turn send your program to the NeWS server.

SEE ALSO

sh(1), say(1)

NeWS Manual

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

psload – display load average under NeWS.

SYNOPSIS

psload [**-u** *update*] [**-h** *history*]

DESCRIPTION

Psload displays a graph in a NeWS window of the system load average. The graph is updated every *update* minutes, and contains information spanning an interval of *history* minutes. *Update* defaults to 0.083 minutes (5 seconds) and *history* defaults to 10 minutes.

SEE ALSO

NeWS Manual

NAME

psman – display reference manual pages; find reference pages by keyword

SYNOPSIS

psman [*section*] *title*
psman **–k** *keyword*

DESCRIPTION

psman displays information from the reference manuals. It can display complete manual pages that you select by *title*. It can display one-line summaries selected by *keyword* (**–k**).

When **–k** is not specified, *psman* formats a specified manual page by *title*. A *section*, when given, applies to the *title* that follows it on the command line. *psman* looks in the indicated section of the manual for the *title*. *section* should be a digit. If *section* is omitted, *psman* searches all reference sections (giving preference to commands over functions) and prints the first manual page it finds. If no manual page is located, *psman* prints an error message.

The following line instructs *psman* to look in section 8 of the reference manual for the **ypwhich(8)** manual page:

```
babylon% psman 8 ypwhich
```

If the NeWS server is not available *psman* formats for a teletype and pipes its output through *more*(1). Otherwise, *psman* formats for a typesetter and pipes its output through *psview*(1).

OPTIONS

–k *keyword* . . .
psman prints out one-line summaries from the *whatis* database (table of contents) that contain any of the given *keyword*s.

ENVIRONMENT

MANPATH If set, its value overrides */usr/man:$NEWSHOME/man* as the default search path.

SEE ALSO

cat(1V), col(1V), eqn(1), more(1), nroff(1), tbl(1), troff(1), whatis(1), man(7), catman(8)

NAME

psterm — NeWS terminal emulator

SYNOPSIS

psterm [options] [command]

DESCRIPTION

psterm is a *termcap*-based terminal emulator program for NeWS. When invoked, it reads the */etc/termcap* entry for the terminal named by the **-t** option, or by the **TERM** environment variable, and arranges to emulate the behavior of that terminal. It forks an instance of *command* (or, by default, the program specified by the **SHELL** environment variable, or *csh* if this is undefined), routing keyboard input to the program and displaying its output.

psterm scales its font to make the number of rows and columns specified in the */etc/termcap* entry for the terminal it is emulating fit the size of its window. It also responds to (most of) the particular escape sequences that *termcap* defines for that terminal.

OPTIONS

−C
: route */dev/console* messages to this window, if supported by the operating system.

−f
: Bring up a reasonably-sized terminal in the lower-left corner of the screen (or in the location specified with the −xy option) instead of having the user define its size and location.

−w
: wait around after the *command* terminates.

−fl *frame label*
: Use the specified string for the frame label.

−il *icon label*
: Use the specified string for the icon label. The icon label normally defaults to the name of the host on which *psterm* is running.

−li *lines* specifies the height of the window in characters.

−co *columns*
: specifies the width of the window in characters.

−xy *x y* specifies the location of the lower left hand corner of the window (in screen pixel coordinates).

−bg
: causes *psterm* to place itself in the background by disassociating itself from the parent process and the controlling terminal. If *psterm* is invoked with *rsh*(1), this option will cause the rsh command to complete immediately, rather than hang around until *psterm* exits.

−ls
: causes *psterm* to invoke the shell as a login shell. In addition, any specified *command* will be passed to the shell with a **−c** option, rather than being invoked directly, so that the shell can establish any environment variables that may be needed by the command. Further, if *psterm* is invoked via *rsh*(1), the host at the other end of the *rsh* socket will be used as the server, unless a NEWSSERVER environment variable is present.

−pm
: specifies that a *psterm* should enable *page mode*. When page mode is enabled and a command produces more lines of output that can fit on the screen at once, *psterm* will stop scrolling, hide the cursor, and wait until the user types a character before resuming output. When *psterm* is blocked with a screenfull of data, typing a carriage return or space will cause scrolling to proceed by one line or one screenful, respectively; any other character will cause the next screenfull to appear and be passed through as normal input. This mode can also be enabled or disabled interactively, using the *Page Mode* menu item.

SELECTION

Clicking the left mouse button over a character selects that character. Clicking it beyond the end of the line selects the newline at the end of that line. Clicking the middle mouse button over a character when a primary selection does not exist in that window selects that character. Clicking the middle mouse button over a character when a primary selection does exist in that window extends or shrinks the selection to that character.

Note that selections are made by **clicking**. Mouse tracking is not implemented yet.

The Copy key (L6) copies the *primary* selection to the *shelf*. The Paste key (L8) copies the contents (*shelf* to the *insertion point* .

If you make a selection while holding down the Copy key, the selection will be a secondary sele Subsequently letting go of the Copy key copies the *secondary* selection to the *shelf* and deselects the s dary selection.

Making a selection while holding down the Paste key also makes a secondary selection. It pastes th *mary* selection to the location of the secondary selection and deselects the secondary selection.

Copy and Paste of both primary and secondary selections work across separate invocations of *psterm.* do not work between *psterm* and SunView. However, a mechanism does exist for transferring a Sur selection to the NeWS shelf, and vice versa. See the description of *Selection Transfer* in *news_server*(1

MENU ITEMS

Psterm adds two items to the top of the standard menu associated with the right hand mouse button. items permit the page mode and automatic margin modes to be turned on and off. Menu items c according to the state of each mode. For example, if page mode is enabled, the menu item will in ''Page Mode Off''.

FILES

/etc/termcap to find the terminal description.

SEE ALSO

news_server(1)

NeWS Manual

BUGS

Emulating some terminal types works better than others, largely because there are incomplete *[illegible]* entries for them.

A large number of *termcap* fields have yet to be implemented.

Page Mode gets easily confused.

NAME

psview – POSTSCRIPT language previewer for NeWS

SYNOPSIS

psview [*POSTSCRIPT language-file*]

DESCRIPTION

Psview puts up a window and runs the user's POSTSCRIPT language code in it. *Psview* uses a portion of the window that has the proper aspect ratio for a standard letter-size page in portrait orientation.

If *POSTSCRIPT language-file* is specified, the POSTSCRIPT language code is taken from that file. If no argument is given, or if a '-' is given as the argument, *psview* reads the POSTSCRIPT program from standard input.

Psview lets you flip through the pages. Page boundaries are determined by locating the %%Page: comments. *Psview* provides a slider to move to any page, and a menu to go to the first, previous, next or last page. Clicking the left mouse button goes to the next page.

SEE ALSO

psh(1), say(1), newsdemos(6)

NeWS Manual

PostScript Language Reference Manual , Adobe Systems Inc., Addison-Wesley

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

BUGS

Assumes a syntactically valid POSTSCRIPT language file.

NAME

say – execute POSTSCRIPT language fragment

SYNOPSIS

say [*options*] [*strings*]

DESCRIPTION

say connects to the NeWS server and displays the strings provided on the command line in a window. An option is provided to interpret the command line, or the standard input, as a POSTSCRIPT program to be executed by the server.

say is used to implement some of the NeWS demo programs. This technique allows window applications to be shell scripts.

OPTIONS

–b*string*
: Use *string* as the title for the window.

–c
: Center the text in the window.

–p
: The command line contains a POSTSCRIPT program rather than simply text strings.

–P
: The standard input contains a POSTSCRIPT program, which is executed after the POSTSCRIPT language commands on the command line (if any).

–r
: Make the window round.

–s*nn*
: Use *nn* as the point size of the text.

–w
: Wait for the window to be destroyed. The default is for the window to vanish when execution of its POSTSCRIPT program is finished.

–W
: Do not create a window for the POSTSCRIPT to be executed in. This can be used to implement operations that do not require a window; for example toggling drag mode in the window manager, or running POSTSCRIPT language code that creates its own window.

–*xxx,yyy*
: The first *xxx,yyy* pair of numbers sets the X and Y coordinates of the window. If a second –*xxx,yyy* command line option is given, it sets the size of the window.

USAGE

Older programs that use *say* solely to send POSTSCRIPT to the NeWS server (by specifying the **-P -w -W " "** options to *say*), should be converted to use *psh (1)*.

SEE ALSO

psh(1), news_server(1)

NeWS Manual

PostScript Language Reference Manual , Adobe Systems Inc, Addison-Wesley

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

setnewshost – generate a string for the NEWSSERVER environment variable

SYNOPSIS

setnewshost *hostname*

DESCRIPTION

setnewshost generates and prints the proper value of the NEWSSERVER environment variable for the given *hostname*. If NEWSSERVER is set then NeWS clients will attempt to connect to the server it points to rather than the local host.

The format of the NEWSSERVER environment variable is as follows:

decimal-address . *port#* ; *hostname*

For example, if the host called ''paper'' has address 192.98.34.118, the NEWSSERVER variable should be set to ''3227656822.2000;paper'' so that NeWS clients will connect to the NeWS server on ''paper''. *setnewshost* simply calculates this string and sends it to standard output. This is not its most convenient form, however. C-shell users can define the following alias:

```
alias snh 'setenv NEWSSERVER `setnewshost \!*`'
```

and System V Bourne Shell users can define the following function:

```
snh () {
   NEWSSERVER=`setnewshost $*`
   export NEWSSERVER
}
```

Both forms let you simply type '**snh** *hostname*' to set the NEWSSERVER environment variable automatically.

SEE ALSO

psh(1)

NeWS Manual

BUGS

The host table entry must have exactly the following format: **a.b.c.d**<*tab*>**hostname**.

If you use the **snh** alias or shell function, and the hostname you give is unknown, or you give too many or too few arguments, the NEWSSERVER variable will be trashed.

NAME

xdemos – X Window System demonstration

SYNOPSIS

X Demos menu item in the NeWS root menu

DESCRIPTION

The NeWS **X Demos** are some programs that demonstrate a small subset of the X Window System (version 10) protocol running under NeWS. The X server is written almost entirely in the POSTSCRIPT language; the code is contained in the file *$NEWSHOME/lib/NeWS/X10.ps.*

The **X Demos** pull-right menu has two items: **Run a Demo** and **Kill X Server**. If the X server is not running, the second menu item will have no effect.

RUNNING DEMOS FROM THE MENU

In order to start an X demo from the NeWS menu, you must pull right from the **Run a Demo** menu item. This reveals a number of demo programs: *Analog Clock and Load Average.* Selecting one of these starts up the corresponding X program. If the X Server has not yet been started, it will be started automatically. The demonstration programs do not bring up windows in the standard NeWS style; they instead conform to the standard X style, which is as follows. First, the name of the program appears in a small window in the upper left-hand corner of the screen. Then, a flickering rectangle or square appears at the current cursor location. This rectangle tracks the motion of the cursor. At this point, you can do one of three things:

Click the left button. This brings up a window of default size at the current cursor location.

Drag with the middle button. If you hold down the middle button, drag the mouse, and then release the middle button, you can drag out a rubber-band rectangle that specifies the window's location and size.

Click the right button. This brings up a window of default size at the default location.

If you select **Kill X Server** while any X demos are active, they will be killed along with the server.

WINDOW MANAGEMENT

In the X Window System, the only way to move or resize windows is to use a window manager program. Unfortunately, no X window managers work under NeWS as yet. In order to provide window management facilities to X windows running under NeWS, a special window management menu is made available in all X windows. This menu is accessible from the right mouse button when the cursor is over an X window and it contains the following items: *Move, Redisplay, Resize,* and *Quit.* These functions should be self-explanatory.

RUNNING DEMOS FROM A SHELL

In order to run X demos from a shell (on the local machine or on a remote machine) one must set the DISPLAY environment variable properly. The NeWS X server listens for connections on port 5901, which means that the DISPLAY variable should contain the string ''hostname:1'' so that X clients will connect to the proper location. After DISPLAY is set properly, one can start up X demos in the normal X fashion.

SEE ALSO

newsdemos(6)
NeWS Manual

BUGS

No byte swapping is done.

Only a small subset of X requests is implemented. In particular, only programs like *xclock* and *xload.*

The NeWS X Server accepts only internet-domain connections. UNIX-domain connections are not supported.

Running the X demos from the menu can freeze the screen and input if you have a line in your .Xde faults file similar to the following: .MakeWindow.Freeze: on Turning on this option causes X clien to issue a slightly different set of requests, two of which the X demo do not handle. The work-around is t change the line in .Xdefaults to: .MakeWindow.Freeze: off

$NEWSHOME/lib/NeWS/demomenu.ps contains an error which causes the X demos to not run some times. The work-around is to replace the following lines in $NEWSHOME/lib/NeWS/demomenu.ps % do "setenv DISPLAY localhost:1" (DISPLAY) (localhost:1) putenv with: % do "setenv DISPLA 'hostname':1" (DISPLAY) localhostname (:1) append putenv

The following is not really a bug as much as an exposure of the NeWS/X interface. X demo windows wi not respond to 'Repaint All' requests from the root menu, or 'Zap All', 'Open All' requests from the 'A Windows' menu. The 'Repaint All' problem is due to a known inadequacy of the current X emulator dem package. 'Zap All' and other selections from the 'All Windows' menu have no effect since X and NeW exist, in essence, in two different worlds.

TRADEMARK

POSTSCRIPT is a registered trademark of Adobe Systems Inc.

NAME

psio – NeWS buffered input/output package

SYNOPSIS

```
#include "psio.h"

PSFILE *psio_stdin;
PSFILE *psio_stdout;
PSFILE *psio_stderr;
```

DESCRIPTION

The functions described here constitute a user-level I/O buffering scheme for use when communicating with NeWS. This package is based on the standard I/O package that comes with Unix. The functions in this package are used in the same way as the similarly named functions in Standard I/O.

The in-line macros *psio_getc* and *psio_putc* handle characters quickly. The higher level routines *psio_read*, *psio_printf*, *psio_fprintf*, *psio_write* all use or act as if they use *psio_getc* and *psio_putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **PSFILE**. *psio_open* creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **psio.h** include file and associated with the standard open files:

psio_stdin standard input file

psio_stdout
standard output file

psio_stderr
standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams.

Any module that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include "psio.h"
```

The functions and constants mentioned in here are declared in that header file and need no further declaration. The constants and the following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc*, *putc*, *psio_eof*, *psio_error*, *psio_fileno*, and *psio_clearerr*.

SEE ALSO

open(2V), close(2), read(2V), write(2V), intro(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), getc(3S), printf(3S), putc(3S), ungetc(3S).

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **PSFILE** pointer has not been initialized with *psio_open*, input (output) has been attempted on an output (input) stream, or a **PSFILE** pointer designates corrupt or otherwise unintelligible **PSFILE** data.

LIST OF FUNCTIONS

Name	*Description*
psio_clearerr	stream status inquiries
psio_close	flush a stream
psio_eof	stream status inquiries
psio_error	stream status inquiries
psio_fdopen	open a stream
psio_flush	close or flush a stream
psio_fileno	stream status inquiries

psio_fprintf	formatted output conversion
psio_getc	get character or integer from stream
psio_open	open a stream
psio_read	buffered binary input/output
psio_printf	formatted output conversion
psio_putc	put character or word on a stream
psio_ungetc	push character back into input stream
psio_write	buffered binary input/output

Index

Special Characters

^C, 131

A

acceptconnection, 131
Action, 19, 121
activate, 131
addeditkeysinterest, 52
addfunctionnamesinterest, 52
addfunctionstringsinterest, 51
addkbdinterests, 51
addselectioninterests, 55
AllEvents, 23, 120
append, 34
append, 34
arccos, 131
arcsin, 131
arctan, 131
array
  ExecutionStack, 125
  Interests, 125
  OperandStack, 125
  StandardErrorNames, 125
  State, 125
arraydelete, 35
arraydelete, 35
arrayinsert, 35
arrayinsert, 35
arrayop, 35
arrayop, 35
autobind, 153
  currentautobind, 154
  setautobind, 154
awaitevent, 132, 13, 21, 22, 134, 137, 144

B

banddevice, 151
bindkey, 44
`bldfamily(1)` — build font family description, 169, 170, 213
blockinputqueue, 132, 26, 132, 147
BottomCanvas, 119
breakpoint, 132, 133, 147
buildimage, 132
buttons
  AdjustButton, 83

buttons, *continued*
  MenuButton, 83
  PointButton, 82
byte stream format, **157**

C

canvas, **14**, 5
  AllEvents, 23, 120
  as **canvas** dictionary, 119
  as canvas object, 117
  BottomCanvas, 119
  CanvasAbove, 119
  CanvasBelow, 119
  canvastobottom, 133
  canvastotop, 133
  clipcanvaspath, 133
  Color, 120
  createdevice, 134
  createoverlay, 134
  currentcanvas, 135
  cursor, 16
  damage, see *damage*
  damagepath, 136
  eoreshapecanvas, 137
  eowritecanvas, 137
  eowritescreen, 137
  EventsConsumed, 15, 23, 120
  framebuffer, 43
  getcanvascursor, 138
  imagecanvas, 140
  imagemaskcanvas, 140
  insertcanvasabove, 140
  insertcanvasbelow, 141
  interest list, 23
  Interests, 121
  Mapped, 14, 120
  mapping, 14
  MatchedEvents, 23, 120
  matching events against, 23
  movecanvas, 142
  newcanvas, 142
  NoEvents, 23, 120
  opaque, 14
  Parent, 119
  readcanvas, 143
  reshapecanvas, 144
  retained, 14
  Retained, 120

canvas, *continued*
  SaveBehind, 120
  setcanvas, 144
  setcanvascursor, 145
  TopCanvas, 119
  TopChild, 119
  transparent, 14
  Transparent, 14, 120
  visibility, 14
  writecanvas, 148
  writescreen, 148
Canvas, 19, 23, 54, 121
  event matching, 22
  in window enter and exit events, 24
CanvasAbove, 119
CanvasBelow, 119
canvastobottom, 133, 14
canvastotop, 133, 14
case comparator, 34
case, 34
`cdef`, 98
/changeitem
  in menu class, 77
charpath, 151
cid utilities, 40
cidinterest, 40
cidinterestlonly, 41
classbegin, 65
classend, 65
classes, **63**, 65
  Object, 66
  ScrollBarItem, 74
  SimpleScrollbar, 74
clearselection, 55
/ClientData, 37, 122
ClientMenu, 78
clipcanvas, 133, 136
clipcanvaspath, 133, 133
clock program, 109
code, see *example code*
color
  `colors.ps`, 32
  as **color** dictionary, 45
  as color object, 117
  contrastswithcurrent, 133
  currentcolor, 135
  hsbcolor, 140
  NeWS' implementation, 7
  rgbcolor, 144
  setcolor, 145
Color, 120
`colors.ps`, 32
`compat.ps`, 33
connection management, 127
  `ps_close_PostScript()`, 127
console, 43, 35
constants, 43
contrastswithcurrent, 16
contacting the server, 178
  for debugging, 87
  interpreter access, 228
contacting the server, *continued*
  multiple servers, 228
  `NEWSSERVER`, 163, 251
  remote access, 228
  `setnewshost(1)` — set NEWSSERVER environm
/ContentsAscii, 54
/ContentsPostScript, 54
continueprocess, 133, 147
contrastswithcurrent, 133
copy events, 19
copyarea, 133, 14, 137
copypage, 151
countinputqueue, 134
cps
  argument types, 99
  `cdef`, 98
  how to use, 97
  parameters, 99
  tagprint, 101
  tags, 101
  the `.c` file, 100
  the `.cps` file, 98
  the `.h` file, 100
  typedprint, 101
cps utilities, 106
  tagprint, 147
  typedprint, 147
  versions for other languages, 163
`cps(1)` \m construct C to POSTSCRIPT language interface,
createcanvas, 35
createcanvas, 35
createdevice, 134, 14
createevent, 134, 19, 132, 137, 144
createmonitor, 134, 141, 142
createoverlay, 134, 37
cshow, 39
currentautobind, 134, 144, 154
currentcanvas, 135
currentcolor, 135
currentcursorlocation, 135, 16
/currentindex
  in menu class, 76
currentinputfocus, 59
/currentkey
  in menu class, 76
currentlinequality, 135, 146
currentpath, 135, 137
currentprintermatch, 135, 146
currentprocess, 135
currentrasteropcode, 135, 147
currentscreen, 151
currentstate, 136, 147
currenttime, 136, 21
currenttransfer, 151
cursor
  building, 32
  canvas, 16
  currentcursorlocation, 16, 135
  cursor control, 52
  `Cursor.ffam`, 167

cursor, *continued*
    `Cursor12.font`, 167
    cursorfont, 32, 38
    font, 16, 32
    fonts, 167
    format, 168
    getcanvascursor, 138
    hot spot, 16
    images, 16
    inheriting, 16
    mask image, 16
    primary image, 16
    representation, 167
    setcanvascursor, 145
    setcursorlocation, 16, 145
    setstandardcursor, 38
    standard font, 167
`Cursor.ffam`, 167
`cursor.ps`, 32, 38
`Cursor12.font`, 167
cvas, 42
cvas, 42
cvis, 42
cvis, 42

D

damage, **14**
    clipcanvas, 133
    damagepath, 136
    eoextenddamage, 138
    extenddamage, 138
    responding to damage, 15
    typical repair sequence, 15
/Damaged, 109
damagepath, 136, 15, 24, 133
dbgbreak, 90
dbgbreakenter, 88
dbgbreakexit, 88
dbgprintf, 34, 90, 91
dbgprintfenter, 89
dbgprintfexit, 89
de-referencing, 128
debugging, **87**
    contacting the server, 87
    dbgbreak, 88
    dbgbreakenter, 90
    dbgbreakexit, 90
    dbgcall, 92
    dgbcallbreak, 92
    dbgcontinue, 91
    dbgcontinuebreak, 91
    dbgcopystack, 92
    dbgenter, 92
    dbgenterbreak, 88, 91
    dbgexit, 92
    dbggetbreak, 92
    dbgkill, 93
    dbgkillbreak, 93
    dbglistbreaks, 88, 90
    dbgmodifyproc, 93
    dgbpatch, 93
    dbgpatchbreak, 92
debugging, *continued*
    dbgprintf, 89
    dbgprintfenter, 90
    dbgprintfexit, 91
    dbgstart, 87, 90
    dbgstop, 90
    dbgwhere, 91
    dbgwherebreak, 91
    `debug.ps`, 33, 87
    errored, 36
    errors, 89
`define_stack_token()`, 105
`ps_define_value_token()`, 105
`ps_define_word_token()`, 106
/deleteitem
    in menu class, 77
demos
    `demomenu.ps`, 32
    `itemdemo`, 189
    newsdemos(6) — demos menu, 231
    `rubber`, 37
    xdemos(6) — X Window System demonstration, 253
/destroy
    in window class, 78
dictbegin, 36
dictend, 36
dictend, 36
dictionary, **183**
    `colors.ps`, 32
    creating your own, 184
    de-referencing, 128
    dictbegin, 36
    magic dictionaries, 117
    new NeWS types, 117
    size of **userdict**, 152
    system, 32, 183
    `systoklst.ps`, 32
    UnloggedEvents, 45
    user, 183
DictionaryStack, 124
distribution mechanism, see *event distribution*
doit, 66
`dumpfont`(1) — dump font in another format, 169, 170, 217
dumpsys, 136

E

echo, 151
emptypath, 136
enumeratefontdicts, 136
environment variables
    getenv, 138
    putenv, 143
eoclipcanvas, 136
eocopyarea, 137
eocurrentpath, 137
eoextenddamage, 138
eoreshapecanvas, 137
eowritecanvas, 137, 148
eowritescreen, 137, 137, 148
error
    accept, 124

error, *continued*
- dictfull, 124
- dictstackoverflow, 124
- dictstackunderflow, 124
- execstackoverflow, 124
- interrupt, 124
- invalidaccess, 124
- invalidexit, 124
- invalidfileaccess, 124
- invalidfont, 124
- invalidrestore, 124
- ioerr, 124
- killprocess, 124
- limitcheck, 124
- nocurrentpoint, 124
- none, 124
- rangecheck, 124
- stackoverflow, 124
- stackunderflow, 124
- syntaxerror, 124
- typecheck, 124
- undefined, 124
- undefinedfilename, 124
- undefinedresult, 124
- unimplemented, 124
- unmatchedmark, 124
- unregistered, 124
- VMerror, 124

error handling, 127

ErrorCode, 124

ErrorDetailLevel, 125

errored, 36

errors, 36

event, 6
- absorption, 23
- as **event** dictionary, 121
- as event object, 117
- **awaitevent**, 132
- **blockinputqueue**, 132
- boundary crossing events, 24
- client-generated events, 19
- consumption, 15
- **copy** and events, 19
- **createevent**, 134
- distribution, 21
- **eventmgrinterest**, 38
- exclusivity, 23
- **expressinterest**, 20, 137
- **geteventlogger**, 139
- ideal, 20
- interest lists, 23
- interest matching, 21
- interests, **20**
- interests and **forkeventmgr**, 37
- **Interests** key in **process**, 125
- keystrokes, 24
- matching, **21**
- order of interest matching, 23
- queue, 21
- **recallevent**, 143
- receiving events, 21
- **redistributeevent**, 143
- **revokeinterest**, 20, 144

event, *continued*
- **sendevent**, 144
- **seteventlogger**, 145
- synchronization, 25
- synthetic events, 19
- types of events, 24

event fields
- **Action**, 19, 21, 121
- **Canvas**, 19, 22, 24, 121
- **ClientData**, 122
- **EventsConsumed**, 15, 23
- **Exclusivity**, 20, 122
- **Interest**, 122
- **IsInterest**, 19, 20, 122
- **IsQueued**, 20, 122
- **KeyState**, 122
- **Name**, 19, 21, 122
- **Priority**, 20, 23, 123
- **Process**, 22, 123
- **Serial**, 20, 123
- **TimeStamp**, 19, 24, 123, 141
- **XLocation**, 19, 123
- **YLocation**, 19, 123

event logging
- `eventlog.ps`, 33

eventlog, 45

eventmgrinterest, 38

events
- /**Damaged**, 24
- /**EnterEvent**, 24
- /**ExitEvent**, 24
- //**LeftMouseButton**, 24
- /**MiddleMouseButton**, 24
- /**MouseDragged**, 24
- /**RightMouseButton**, 24
- /**DownTransition**, 24
- logging, 26, 45
- mouse, 24
- timer, 24
- unlogging, 45
- /**UpTransition**, 24
- window crossings, 24

EventsConsumed, 15, 23, 120

example code
- interactive NeWS server session using `psh`, 178
- `lines`, 80
- periodic events, 27
- `roundclock`, 109
- setting cursor shape, 38
- using the window and menu packages, 80

example programs
- rubber-banding, 37
- `test.psh`, 180

Exclusivity, 20, 23, 122

Execee, 125

executeonly, 151

ExecutionStack, 125

executive, 87, 178

expressinterest, 137, 20, 21, 132, 134, 143, 144

extenddamage, 138

F

file
  raster, 143
file, 138
fillcanvas, 39
findfilefont, 42
findfilefont, 42
findfont
  Building NeWS fonts, 170
/flipiconic
  in window class, 79
font, **167**
  vfont format, 170
  ASCII version, 168
  binary version, 169
  `bldfamily(1)` — build font family description, 169, 213
  building ordinary fonts, 170
  `Cursor.ffam`, 167
  `Cursor12.font`, 167
  dictionary generation, 169
  directory, 169
  `dumpfont(1)` — dump font in another format, 169, 217
  family, 170
  generation, 167
  `iconedit`, 168
  iconfont, 32
  `mkiconfont`, 168
  representation, 167
font object, 42
font utilities, 41
  cvas, 42
  cvis, 42
  findfilefont, 42
  fontascent, 41
  fontdescent, 42
  fontheight, 41
  stringbox, 42
fontascent, 41
fontascent, 41
fontdescent, 42
fontdescent, 42
fontheight, 41
fontheight, 41
fork, 147
forkeventmgr, 37
forkitems, 190
ForkPaintClient?, 79
forkunix, **138**, 32
format
  byte stream, 157
fprintf, 35
fprintf, 35, 43
framebuffer, 43
framedevice, 151
FrameLabel, 78, 79
function keys
  assigning, 43
  repeating, 44

G

garbage collection, 128
  de-referencing, 128
get, 184
getanimated, 36
getcanvascursor, 138, 16, 145
getcanvaslocation, 138, 142
getclick, 37
getenv, 138
geteventlogger, 139, 145
getkeyboardtranslation, 139
getkeyboardtranslation, 139, 141
getmousetranslation, 139
getrect, 37
getselection, 56
getsocketlocaladdress, 139
getsocketpeername, 139
getvalue, 34
getvalue, 34
getwholerect, 37
global interest list, 21, 23
globalinterest, 140, 21
graphics utilities, 39
graphicsstate
  as **graphicsstate** dictionary, 123
  as graphicsstate object, 117
  currentstate, 136
  setstate, 147

H

hasfocus, 60
hsbcolor, 140

I

icon
  fonts, 167
  `icon.ps`, 32
  iconfont, 32
`iconedit`, 168
IconImage, 78
IconLabel, 78
image
  buildimage, 132
  imagecanvas, 140
  imagemaskcanvas, 140
imagecanvas, 140, 15
imagemaskcanvas, 140
implementation limits, 152
`init.ps`
  size of **userdict**, 152
`init.ps`, 31, 82, 174, 183
initclip, 144
PaintRoot, 32
input, **19**
  addeditkeysinterest, 52
  addfunctionnamesinterest, 52
  addfunctionstringsinterest, 51
  addkbdinterests, 51
  extended input system, **49**
  lasteventtime, 141

input, *continued*
  revokekdbinterests, 51
input events, **19**, see also *Event Fields*
input focus, 59
  currentinputfocus, 59
  hasfocus, 60
  setfocusmode, 60
  setinputfocus, 59
insertcanvasabove, **140**, 14
insertcanvasbelow, **141**, 14
/insertitem
  in menu class, 76
insetrect, **39**
insetrrect, **40**
interactive
  example, 178
  server, 178
interest matching
  order of, 23
Interest, 122
interests, see *events*
  globalinterestlist, 140
Interests, **125**
Interests, 121
interpreter access, 228
introduction
  NeWS I/O library functions, 255
invertmatrix, 151
IsInterest, 20, 122
IsQueued, 122
item, **189**
  forkitems, 190
  `liteitem.ps`, 33
  paintitems, 190
`itemdemo`, 189

J

journalend, **42**
journalling, 42
  begin, 42
  controls, 43
  endjournal, 42
  internal variables, 43
  journal, 42
  `journal.ps`, 33
  record, 42
  replay, 42
  stop, 42
journalling(1) — record and playback package, 219
journalplay, **42**
journalrecord, **42**

K

`kbd_mode(1)` — change keyboard translation mode, 221
  NeWS and Sunview, 176
key mapping, 43
keyboard
  function key assignment, 52
  keyboard editing, 52
  repeating keys, 44
keyboard input
keyboard input, *continued*
  ASCII, 50
  function keys, 51, 52
  reset, 221
keyboardtype, **141**, 139, 145
keys
  bindkey, 44
  mapping, 43
  `repeat.ps`, 33
  repeating, 44
  unbindkey, 44
KeyState, 122
killprocess, **141**, 131
killprocessgroup, **141**, 131

L

lasteventtime, **141**, 21
library functions
  introduction to NeWS standard I/O, 255
lightweight processes, see *processes*
line quality
  currentlinequality, 135
  setlinequality, 146
`lines` example code, 80
LiteItem, see *item*
`liteitem.ps`, 33, 189
LiteMenu, see *menu*
`litemenu.ps`, 32, 73
`litetext.ps`, 33
liteUI user interface package, 50
`liteUI.ps`, 32, 49, 122
`litewin.ps`, 32, 36
LiteWindow, see *window*
`litewindow.ps`, 73
localhostname, **141**
logging, 45
logging events, 26

M

magic numbers, 159
/map
  in window class, 78
Mapped, **120**
MatchedEvents, 23, 120
max, **141**
menu
  /changeitem, 77
  /currentindex, 76
  /currentkey, 76
  /deleteitem, 77
  example program `test.psh`, 180
  /insertitem, 76
  keys, 75
  `litemenu.ps`, 32
  modifying the root menu, 184
  /new, 75
  /popup, 76
  root menu, 223
  /searchaction, 76
  /searchkey, 76

menu, *continued*
  /showat, 76
menu keys, 75
menu methods, 74
min, 141
missing POSTSCRIPT language operators, 151
`mkiconfont`, 168
modifying the server, 183
  changing defaults, 185
  changing menus, 184
  changing the window user interface, 83
  saving keystrokes, 93, 184
  `startup.ps`, 32
  `user.ps`, 32, 183
modifyproc, 35
modifyproc, 35
monitor
  as monitor object, 118
  createmonitor, 134
monitor, 141, 134, 142
monitorlocked, 142, 134, 141
mouse, see *cursor*
/move
  in window class, 79
movecanvas, 142, 14, 138
multiple servers, 228

N

Name, 19, 122
new, 66, 73, 191
  in LiteItem class, 191
  in menu class, 75
  in window class, 77
new#, 184
newcanvas, 142, 14
newprocessgroup, 142, 141
NeWS buffered I/O library functions, introduction to, 255
NeWS I/O library functions, introduction to, 255
NeWS protocol, 157
NeWS type extensions
  canvas, 117
  color, 117
  event, 117
  graphicsstate, 117
  monitor, 118
  process, 118
  shape, 118
`news_server(1)` — NeWS server program, 31, 174, 223
  debugging server, 174
  three binaries, 174
newsdemos(6) — NeWS demonstrations, 231
newshost(1) — NeWS network security control., 237
NEWSSERVER, 163, 179
noaccess, 151
NoEvents, 23, 120
nulldevice, 151
nulldict, 43
nullproc, 43
nullstring, 43

O

object tables, 158
Object, 66
objects
  cleanup, 126
  connection management, 127
  error handling, 127
  management, 127
  process management, 127
  server function, 126
offscreen images, 15
omitted POSTSCRIPT language operators, 151
`on(1)`, 179
online reference using NeWS — `man`, 243
opaque canvases, 14
OperandStack, 125
operators, 205
ovalframe, 40
ovalpath, 40
overlay canvas
  createoverlay, 134
  getanimated, 37
`overview(1)`, 174, 175, 223

P

POSTSCRIPT language files loaded by the server, see `ps` *files*
packages, **73**, 63
  interface description format, 74
/paint, 190
  in window class, 79
/paintclient
  in window class, 79
PaintClient, 78, 178
/paintframe
  in window class, 79
/painticon
  in window class, 79
paintitems, 190
Parent, 119
pathforallvec, 142
pause, 143, 13, 81
pointinpath, 143
points2rect, 39
/popup
  in menu class, 76
portability
  retained, 142
POSTSCRIPT language extensions
  in `*.ps` files, 33
`pprintf()`, 163
previewing POSTSCRIPT language graphics, 178
printer compatability, 32
printername, 32
printf, 34
printf, 34, 67, 89, 194
Priority, 20, 23, 123
process
  as object, 118
  as **process** dictionary, 124
  scheduling policy, 13

process keys
  DictionaryStack, 124
  ErrorCode, 124
  Execee, 125
  ExecutionStack, 125
  Interests, 125
  OperandStack, 125
  State, 125
process management, 127
  garbage collection, 128
  killing an application, 127
Process, 123
  event matching, 22
processes, **13**
  activate, 131
  breakpoint, 132
  continueprocess, 133
  currentprocess, 135
  forking, 74
  forkunix, 138
  killprocess, 141
  killprocessgroup, 141
  newprocessgroup, 142
  pause, 143
  suspendprocess, 147
  waitprocess, 147
prompt, 151
*.ps files, **31**
  colors.ps, 32
  compat.ps, 33
  cursor.ps, 32, 38, 167
  debug.ps, 33, 87
  demomenu.ps, 32
  eventlog.ps, 33
  file location, 31
  icon.ps, 32
  init.ps, 31, 82, 152, 168, 174, 183
  journal.ps, 33
  liteitem.ps, 33, 189
  litemenu.ps, 32, 73
  litetext.ps, 33
  liteUI.ps, 32, 49, 122
  litewin.ps, 32, 36
  litewindow.ps, 73
  menu.ps, 168
  organization, 31
  POSTSCRIPT language procedures they define, 33
  repeat.ps, 33
  startup.ps, 32, 151, 183
  statusdict.ps, 32
  systoklst.ps, 32
  user.ps, 32, 83, 87, 93, 151, 183
  util.ps, 32
ps_define_stack_token(), 105
ps_define_value_token(), 105
ps_define_word_token(), 106
pscanf(), 164
psh(1) — NeWS PostScript shell, **177**, 81, 87, 178, 239
  example program test.psh, 180
psio(3) — NeWS buffered input/output package, 255
psload(1) — display load average under NeWS, 241
man — online display of reference pages using NeWS, 243
psterm(1) — NeWS terminal emulator, 180, 225, 2[illegible]
psview(1) — POSTSCRIPT language previewer, 178, [illegible]
put, 184
putenv, 143

R

random, 143
Rank, 54
raster files, 143
rasteropcode
  currentrasteropcode, 135
  setrasteropcode, 146
readcanvas, 143
recallevent, 143, 20, 144
rect, 38
rect2points, 39
rectangle utilities, 38
rectframe, 40
rectpath, 39
rectsoverlap, 39
redistributeevent, 143, 20, 21, 25, 26, 132, 134, 137, 144
remote access, 228
renderbands, 151
resetfile, 151
/reshape
  in window class, 78
reshapecanvas, 144, 14, 137, 142
/reshapefromuser
  in window class, 78
restore, 151
retained
  portability, 142
retained canvases, 14
Retained, 120
reversepath, 151
revokeinterest, 144, 20, 137
revokekbdinterests, 51
rgbcolor, 144
roundclock example code, 109
rrectframe, 40
rrectpath, 40
rshow, 39
rubber, 37

S

save, 151
SaveBehind, 120
say(1) — execute POSTSCRIPT language, 177, 249
scalefont
  building NeWS fonts, 170
scene, 5
scrollbars, 74
/searchaction
  in menu class, 76
/searchkey
  in menu class, 76
security
  implementation, 128
selection events, 56

selection events, *continued*
- **DeSelect**, 58
- **ExtendSelectionTo**, 57
- **SelectionRequest**, 58
- **/SetSelectionAt**, 56
- **ShelveSelection**, 58

selection library procedures, 54
/SelectionHolder, 54
/SelectionLastIndex, 54
/SelectionObjsize, 54
selectionrequest, 55
SelectionRequester, 54
/SelectionResponder, 54
selectionresponse, 55
selections, 53
- **addselectioninterests**, 55
- **clearselection**, 55
- **getselection**, 56
- **selectionrequest**, 55
- **selectionresponse**, 55
- **setselection**, 55

/SelectionStartIndex, 54
self, 64
self, 66
send, 65, 64
sendcidevent, 41
sendevent, 144, 20, 21, 132, 134, 137, 143
Serial, 123
server function, 126
setautobind, 144, 134, 154
setcanvas, 144
setcanvascursor, 145, 16, 38, 138
setcolor, 145
setcursorlocation, 145, 16
seteventlogger, 145
setfileinputtoken, 145
setfileinputtoken, 145
setfocusmode, 60
sethsbcolor, 16, 145
setinputfocus, 59
setkeyboardtranslation, 145
setkeyboardtranslation, 145, 139, 141
setlinequality, 146, 135
setmousetranslation, 146
`setnewshost(1)` — set NEWSSERVER environment, 163, 179, 251
setpath, 146
setprintermatch, 146, 135
setrasteropcode, 146, 135
setrgbcolor, 16, 145
setsccbatch, 32
setscreen, 151
setselection, 55
setstandardcursor, 38
setstate, 147, 136
settransfer, 151
setvalue, 34
setvalue, 34
shape
shape, *continued*
- as **shape** dictionary, 126
- as shape object, 118

/showat
- in menu class, 76

showpage, 151
sleep, 36
sleep, 36
sockets
- **acceptconnection**, 131
- **getsocketlocaladdress**, 139
- **getsocketpeername**, 139
- priority of connection, 228

sprintf, 34
sprintf, 34
StandardErrorNames, 125
start, 151
starting NeWS, 174
startkeyboardandmouse, 147
`startup.ps`, 32, 183
State, 125
statusdict.ps, 32
store, 184
stringbbox, 42
stringbox, 42
stroke
- **currentlinequality**, 135
- **setlinequality**, 146

strokecanvas, 39
SunView
- keyboard state if NeWS crashes, 221

SunView1
- and NeWS on the Sun-3/110, 176
- coexistence with NeWS, 174

super, 64
super, 66
suspendprocess, 147, 13, 133
`systoklst.ps`, 32

T

tagprint, 103
tagprint, 147
tags, 101
text
- display using `say(1)`, 177

text utilities, 41
time values, 19
- event queue, 21
- resolution, 123

TimeStamp, 19, 21, 24, 123, 141
/tobottom
- in class window, 80

tokenization, 105
tokens, 105
TopCanvas, 119
TopChild, 119
/totop
- in window class, 80

translate, 151
transparent canvases, 14

Transparent, 120
type extensions, **117**
typedprint, 103
typedprint, 147

U

unbindkey, 44
unblockinputqueue, 147, 26, 132
undef, 147
uniquecid, 40
UnknownRequest, 55
unlogging, 45
/unmap
  in window class, 79
unregistered
  error accessing process keys, 124
user interface
  default, 82
  window management, 227
`user.ps`, 32, 83, 87, 93, 183
usertime, 151
`usertoken`, 105
utilities
  cps, 106
  font, 41
  misc., 33
  text, 41
  `util.ps`, 32

V

verbose?, 32

W

waitprocess, 147, 37, 131
window
  /destroy, 78
  example program `test.psh`, 180
  /flipiconic, 79
  ForkPaintClient?, 79
  `litewin.ps`, 32
  /map, 78
  /move, 79
  /new, 77
  /paint, 79
  /paintclient, 79
  /paintframe, 79
  /painticon, 79
  /reshape, 78
  /reshapefromuser, 78
  scrollbars, 74
  /tobottom, 80
  /totop, 80
  /unmap, 79
window management, 227
window methods, 77
writecanvas, 148, 137
writescreen, 148, 137, 148

X

xdemos(6) — X Window System demonstration, 253
XLocation, 19, 123

Y

YLocation, 19, 123

Revision History

Version	Date	Comments
A	29 March 1987	First release of the *NeWS Manual*.
50	2 October 1987	First release of the β version of the *NeWS Manual*.
A	15 January 1988	FCS 1.1 release of the *NeWS Manual*.