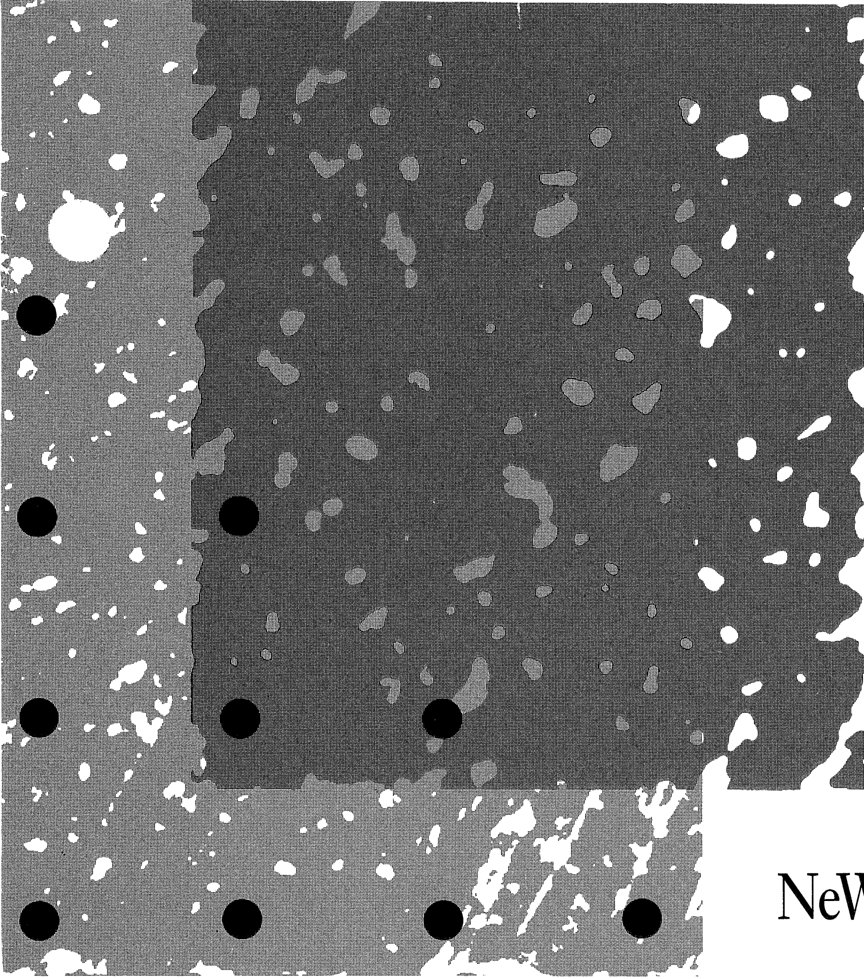
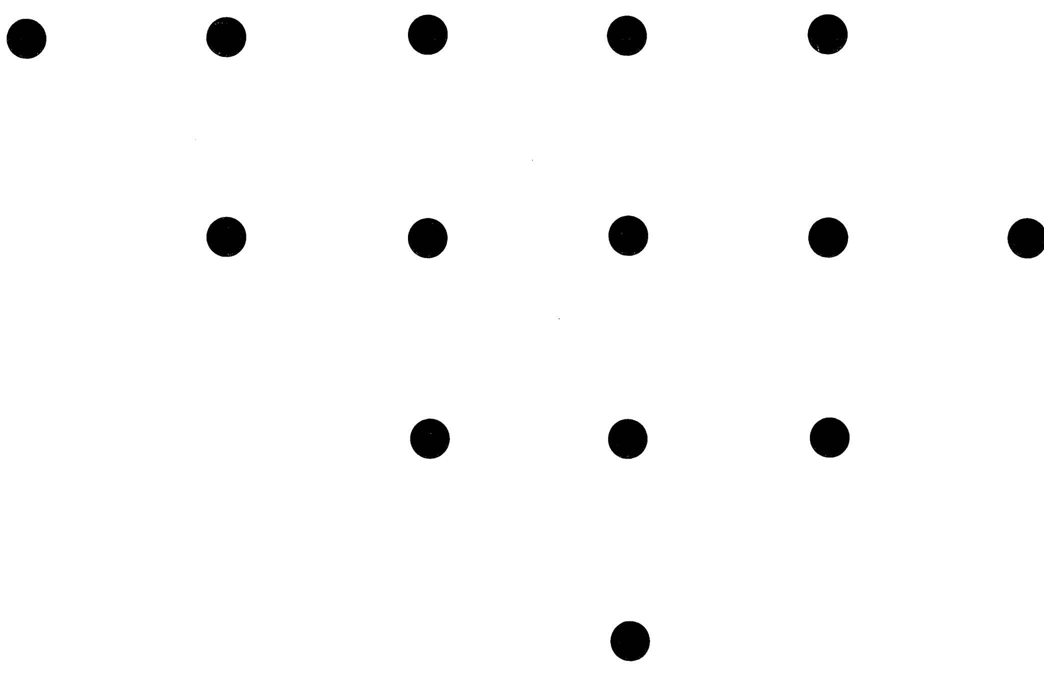


NeWS 2.1 Programmer's Guide



Part Number: 800-4888-10
Revision A, of 11 June 1990

SunView, XView, and OpenWindows are trademarks of Sun Microsystems, Inc.

NeWS, 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.

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

UNIX is a registered trademark of AT&T.
OPEN LOOK is a trademark of AT&T.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Copyright © 1990 Sun Microsystems, Inc. – Printed in U.S.A.

All rights reserved. No part of this work covered by copyright hereon may be reproduced in any form or by any means – graphic, electronic, or mechanical – including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

Restricted rights legend: use, duplication, or disclosure by the U.S. government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

Contents

Chapter 1 Introduction **1**

1.1. NeWS Programming: An Overview 1

The POSTSCRIPT Language 1

NeWS Types 1

NeWS Operators 2

The X11/NeWS Server 2

NeWS Processes 2

Client-Server Communication 2

C Client Interface 3

Canvases 3

Imaging Model 4

Events 4

Memory Management 4

Color Support 5

Font Support 5

Multiple Screen Support 5

1.2. POSTSCRIPT Language Files Used with the Server 6

Classes 6

Debugging 6

Utilities 6

Chapter 2 Canvases **7**

The canvastype Extension 7

Canvas Operators 8

2.1. Basic Terms and Concepts 8
- Using Multiple Canvases to Create a Window 9
- The Canvas Hierarchy 10
- Opaque and Transparent Canvases 12
- Visibility and Mapping 13
- Canvas Damage 13
- Retained and Unretained Canvases 13
- Rooted and Unrooted Canvases 14
- Coordinate Systems 14

2.2. Basic Canvas Operations 15
- Creating Canvases 15
- Setting a Canvas' Shape and Coordinate System 16
- Mapping Canvases to the Screen 17
- Setting the Current Canvas 18
- Drawing on Canvases 18
- Moving Canvases 20
- Getting the Location of a Canvas 22
- Destroying Canvases 22

2.3. Using the Transparent and Opaque Properties of Canvases 22
- Opaqueness and Transparency 23
- Painting on a Transparent Canvas 25
- Making a Parent Canvas Transparent 27

2.4. Canvas Damage: When to Expect It, How to Fix It, How to Avoid It 28
- When is a Canvas Damaged? 28
- Repairing Canvas Damage 29
- Avoiding Canvas Damage with Retained Canvases 29
- Avoiding Canvas Damage with SaveBehind Canvases 32

2.5. Restricting the Drawing Area with the Canvas Clip 33

2.6. Manipulating the Canvas Hierarchy 35
- Changing Sibling Relationships 36
- Establishing a New Parent 39

2.7. Overlay Canvases 40

Creating and Using Overlays 41
Restrictions for Drawing on Overlays 44
The Framebuffer Overlay 44
2.8. Canvases, Files, and Imaging Procedures 44
Writing Canvases to Files 45
Reading Canvases from Files 47
Imaging a Canvas to the Screen 47
Building a Canvas Image 49
2.9. Cursors 51
Cursor Objects 51
Standard Cursors 52
Changing a Canvas' Cursor 53
Changing a Cursor's Color 54
2.10. Using Multiple Screens 54
The Global Root Canvas 55
Creating the Framebuffer Canvases 55
Allowed Operations for the Global Root Canvas 55
Benign Operations for the Global Root Canvas 56
Disallowed Operations for the Global Root Canvas 56

Chapter 3 Processes **57**
The **processtype** Extension 58
Process Operators 58
3.1. Basic Process Operations 59
Establishing a Client Connection Process 59
Returning the Current Process 60
Examining the Process Stacks 60
Operand Stack 60
Execution Stack 61
Dictionary Stack 61
Graphics State Stack 61
Creating a New Process 62
Process Scheduling: Allowing Other Processes to Run 63

Pausing 64

Waiting 65

Sleeping 66

Examining the Process Execution State 67

Destroying Processes 67

Suspending and Restarting Processes 69

Using the `psps` Utility 70

3.2. Creating and Manipulating Process Groups 70

3.3. Using Monitors for Synchronization 73

3.4. Handling Errors 76

3.5. Controlling Dictionary Sharing Between Parent and Child Processes 77

Chapter 4 Events **79**

The **eventtype** Extension 79

Event Operators 80

4.1. Overview of Event Distribution 80

4.2. Basic Event Operations 83

Creating an Event 83

Expressing Interests 84

Copying an Event Before Expressing Interest 84

Changing and Reusing Interests 84

Sending an Event into Distribution 85

Awaiting Events 85

Using Arrays in an Interest's **Name, Action,** or **Canvas** Key 86

Setting and Inspecting an Event's Location 87

Specifying the Time of an Event's Distribution 89

Specifying Additional Event Information 91

Recalling Events and Revoking Interests 91

4.3. Rules for Matching Events to Interests 91

Rules for Matching **Name** And **Action** Key Values 91

Rules for Matching **Process** Key Values 92

Rules for Matching **Serial** Key Values 92

4.4. Post-Match Processing: Specifying a Dictionary for an Interest's **Name**, **Action**, or **Canvas** Key 92
Specifying Non-Executable Dictionary Values 92
Specifying Executable Dictionary Values 94
4.5. Event Distribution: Matching an Event to Multiple Interests 96
Canvas Interest Lists 96
Pre-Child and Post-Child Interests 96
Assigning Interests to Canvas Interest Lists 96
Interest List Order 97
Order of Interest Matching: Searching the Canvas Hierarchy 97
If the Event's **Canvas** Value is a Single Canvas 97
If the Event's **Canvas** Value is an Array or Dictionary 98
If the Event's **Canvas** Value is **null** 98
Search Path Example 98
Stopping the Search 99
Canvas Event Consumption 99
Exclusive Interests 100
Redistributing an Event Stopped by an Exclusive Interest 100
Modified Search Path Example 101
Hints for Using Pre-Child and Post-Child Interests 102
Example: Matching Multiple Interests 102
4.6. System-Generated Events 109
Keyboard Events 109
Obsolescence Events 109
ProcessDied Events 109
Mouse Events 110
Enter and Exit Events 113
Focus Events 118
Damage Events 120
4.7. Synchronizing Input with Multiple Processes 122
Blocking the Global Event Queue with **blockinputqueue** 122
Blocking the Global Event Queue with the **Synchronous** Key 124
Synchronizing All Events for a Process 126

4.8. Restricting Distribution of an Event to a Specific Process 126
4.9. Creating an Event-Logger Process 128

Chapter 5 Classes **131**

5.1. Basic Terms and Concepts 131
Classes and Instances 131
Instance Variables, Class Variables, and Methods 132
Inheritance and the Class Tree 133
Superclasses and Subclasses 133
The Immediate Superclass 133
Inheritance 134
Single Inheritance and Multiple Inheritance 134
The Inheritance Array 135
A Single Inheritance Example 135
Summary of Terms 137
5.2. Creating a New Class 139
The Class Definition 139
classbegin 139
classend 139
redef 140
Initializing a New Class 140
5.3. Sending Messages With the **send** Operator 140
The Usual Form of **send** 140
The Steps Involved in a **send** 141
Using **send** to Invoke a Method 141
A Nested **send** 142
Using **send** to Create a New Instance 143
Another Form of **send** 144
Using **send** to Change the Value of an Instance Variable 144
Using **send** to Change the Value of a Class Variable 145
5.4. The Pseudo-Variables **self** and **super** 145
The **self** Pseudo-Variable 147
The **super** Pseudo-Variable 148

Using **super** to Send a Message Up the Superclass Chain 150
Restrictions on the Use of **self** and **super** 150
5.5. Method Compilation 150
Compiling **self send** 151
Compiling **super send** 151
Local Dictionaries 151
Controlling Method Compilation 152
/methodcompile 152
/installmethod 153
/doit 153
SetLocalDicts 154
5.6. Creating a New Instance 156
/new 157
/newobject 157
/newinit 158
/newmagic 159
5.7. Intrinsic Classes and Default Classes 161
The **DefaultClass** Variable 161
/newdefault 161
/defaultclass 162
/SubClassResponsibility 162
5.8. Overriding Class Variables With **UserProfile** 162
Overriding **DefaultClass** 163
5.9. Promoting Class Variables to Instance Variables 163
promote 163
unpromote 164
promoted? 164
Avoiding an Accidental Promotion 164
5.10. Destroying Classes and Instances 165
/destroy 165
/destroydependent 165
classdestroy 165
/cleanoutclass 165

5.11. Obsolete Objects in the Class System 165

/obsolete 166

5.12. Multiple Inheritance 166

A Simple Multiple Inheritance Example: a Utility Class 167

A More Complex Multiple Inheritance Example 169

Rules for Valid Inheritance Array Orders 170

Possible Inheritance Arrays for this Example 170

Which Order Do You Choose? 172

Constraining the Order of the Inheritance Array 173

super and Multiple Inheritance 173

5.13. Utilities for Setting and Retrieving an Object's **Name** and **Classname** 174

/name 174

/setname 174

/named? 175

/classname 175

5.14. Utilities for Inquiring About an Object's Status 175

isobject? 175

isclass? 175

isinstance? 175

5.15. Utilities for Inquiring About an Object's Heritage 175

/superclasses 175

/subclasses 176

/instanceof? 176

/descendantof? 176

/understands? 176

/class 176

5.16. Utilities for Finding Objects on the **send** Stack 176

/topmostinstance 176

/topmostdescendant 176

/sendtopmost 177

5.17. Syntax Summary for Class Operators 177

5.18. Syntax Summary for Class Methods 177

Chapter 6 C Client Interface **179**

6.1. The Three Parts of a CPS Client 180

Creating the `.cps` File 180

Creating the `.h` File 180

Creating and Compiling the `.c` File 180

Including Other Header Files in the `.c` and `.cps` Files 181

6.2. CPS Connection Utilities 181

Establishing a Connection 181

Flushing the Output Buffer 182

Closing the Connection 182

Connection Example 182

6.3. The `cdef` Command 183

The `cdef` Syntax 183

CPS Argument Types 184

The Three Types of `cdef` Macros 185

Sending POSTSCRIPT Language Code without Returning Values 186

Receiving Synchronous Replies 186

Receiving Asynchronous Replies 188

6.4. CPS Utilities for Retrieving Input from the Input Connection File 190

6.5. CPS Utilities for Common POSTSCRIPT Language Operators 191

6.6. Defining User Tokens for Efficient Communication 191

NeWS Operators for Manipulating the User Token List 193

Using **setfileinputtoken** to Define a User Token 194

Example 194

Using CPS Utilities to Define a User Token 195

Declaring the User Token 195

Defining the Token's Value 195

Examples 196

6.7. Debugging CPS Clients 197

6.8. An Example CPS Client: The Lunar Lander Game 198

Splitting the Code Between Client and Server 199

The Lunar Lander `.c` File 199
The Lunar Lander `.cps` File 204
The Lunar Lander `lunartags.h` File 208
Program Overview 208
Set-Up and Connection to the Server 208
Initialization 208
The Main Control Loop 209
Clean-Up 210
Making the Shards: A Special Type of `cdef` 210
6.9. Creating an Interface for Clients Not Written in C 211
Hints for Creating a Facility Equivalent to CPS 211
Contacting the Server 211

Chapter 7 Debugging **213**
7.1. Loading the Debugger 213
7.2. Starting the Debugger 213
7.3. Using the Debugger 213
Multi-Process Debugging 214
7.4. Client Commands 214
7.5. User Commands 215
7.6. Debugging Hints 219
Using Aliases 219
Using Multiple Debugging Connections 220

Chapter 8 Memory Management **221**
8.1. Reference Counting 221
Counted and Uncounted Objects 221
References to Counted Objects 222
Counted References 222
Uncounted References 222
Soft References and Obsolescence Events 223
Hard References 223
Reference Tallies 224

8.2. Memory Management Operators 224
Softening a Reference 224
Hardening a Reference 224
Determining a Reference's Type 225
8.3. Memory Management Debugging Operators 225
Counting the Number of Server Objects 225
Returning an Object's Reference Count 226
Printing Information on All Current References 227
Inspecting Memory Usage 228
8.4. Memory Management Debugging Tools 228
8.5. Hints for Debugging Memory Leaks 229
Identifying a Memory Leak 229
Gathering Data 230
Filing Bug Reports 230
8.6. The Unused Font Cache 231
Setting and Inspecting the Size of the Cache 231
Flushing the cache 232
Applications 232

Chapter 9 NeWS Type Extensions **233**
9.1. NeWS Objects as Dictionaries 233
9.2. List of NeWS Types 234
POSTSCRIPT Language Types 234
NeWS Type Extensions 235
9.3. **colortype** 235
9.4. **graphicsstatetype** 235
9.5. **monitortype** 235
9.6. **packedarraytype** 236
9.7. **pathtype** 236
9.8. **canvastype** 236
9.9. **colormaptype** 242
9.10. **colormapentrytype** 242
9.11. **cursortype** 243

9.12. **environmenttype** 244

9.13. **eventtype** 246

9.14. **fonttype** 249

9.15. **processtype** 250

9.16. **visualtype** 255

Chapter 10 NeWS Operator Extensions **257**

Chapter 11 Extensibility through NeWS Procedure Files **291**

11.1. Initialization Files 291

`init.ps` 291

`redbook.ps` 291

`basics.ps` 291

`cursor.ps` 291

`statdict.ps` 291

`compat.ps` 292

`util.ps` 292

`class.ps` 292

11.2. User-Created Extension Files 292

`.startup.ps` 292

`.user.ps` 292

Other Extension Files 292

`debug.ps` 292

`eventlog.ps` 292

`journal.ps` 292

`repeat.ps` 292

Extension File Contents 292

11.3. Miscellaneous Utilities 293

11.4. Array Utilities 297

11.5. Conditional Utilities 299

11.6. Input Utilities 300

11.7. Rectangle Utilities 303

11.8. Graphics Utilities 303
11.9. File Access Utilities 305
11.10. CID Utilities 306
11.11. Journalling Utilities 307
Journalling Internal Variables 307
11.12. Constants 308
11.13. Key Mapping Utilities 309
11.14. Repeating Keys 310
11.15. Logging Events 310
UnloggedEvents 311

Appendix A NeWS Operators **313**
A.1. NeWS Operators, Alphabetically 313
A.2. NeWS Operators, by Functionality 316
Canvas Operators 316
Event Operators 317
Mathematical Operators 318
Process Operators 318
Path Operators 318
File Operators 319
Color Operators 319
Keyboard and Mouse Operators 319
Cursor Operators 320
Font Operators 320
Miscellaneous Operators 320

Appendix B Byte Stream Format **323**
B.1. Encoding For Compressed Tokens 323
B.2. Token Lists 325
System Token List 326
User Token List 326
B.3. Encoding Example 326
B.4. Sending Tagged Replies from Server to Client 327

Appendix C The Extended Input System **329**

C.1. Building on NeWS Input Facilities 329

C.2. The LiteUI Interface 330

C.3. Keyboard Input 331

Keyboard Input: Simple ASCII Characters 331

Revoking Interest in Keyboard Events 331

Keyboard Input: Function Keys 331

Assigning Function Keys 332

Keyboard Input: Editing and Cursor Control 332

C.4. Selections 333

Selection Data Structures 333

Selection Procedures 335

Selection Events 336

/SetSelectionAt 337

/ExtendSelectionTo 338

/DeSelect 339

/ShelveSelection 339

/SelectionRequest 340

C.5. Input Focus 340

Appendix D Omissions and Implementation Limits **345**

D.1. Operator Omissions and Limitations 345

D.2. Font Dictionary Limitations 345

D.3. The **statusdict** Dictionary 345

D.4. Implementation Limits 346

D.5. Other Differences with the POSTSCRIPT Language 347

Index **349**

Tables

Table 4-1 Action Values for Enter and Exit Events 114
Table 4-2 Action Values for Keyboard Focus Events 119

Table 5-1 Summary of Terms 138

Table 6-1 CPS Argument Types 185
Table 6-2 CPS Utilities for POSTSCRIPT Language Operators 191

Table 8-1 Uncounted Object Types 222
Table 8-2 Counted Object Types 222

Table 9-1 Standard Object Types in the POSTSCRIPT Language 234
Table 9-2 Additional NeWS Object Types 235

Table 10-1 Events sent to incanvas and its parents 277
Table 10-2 Events sent to outcanvas and its parents 277
Table 10-3 Rasterop Code Values 285

Table 11-1 Standard NeWS Cursors 302

Table B-1 Token names and their associated values (in octal format) 324
Table B-2 Bytes for binary encoding example (given in octal format) 327

Table C-1 Selection-Dict Keys 334
Table C-2 System-defined Selection Attributes 334

Table C-3 Request-dict Entries 335
Table C-4 High-Level Selection-Related Events 335
Table C-5 Input Focus 341

Table D-1 Implementation Limits 346
Table D-2 NeWS Versions of Various POSTSCRIPT Language Operators 347

Figures

Figure 2-1 A simple window 9

Figure 2-2 Two overlapping windows 11

Figure 2-3 Example hierarchy 11

Figure 2-4 FirstCanvas mapped and filled with gray 19

Figure 2-5 FirstCanvas with star 20

Figure 2-6 FirstCanvas with text string 20

Figure 2-7 FirstCanvas moved to 25, 25 in framebuffer canvas' coordinate system 21

Figure 2-8 FirstCanvas and its child, SecondCanvas 24

Figure 2-9 Newly painted FirstCanvas beneath opaque SecondCanvas 24

Figure 2-10 FirstCanvas beneath transparent SecondCanvas 25

Figure 2-11 Newly painted FirstCanvas beneath transparent SecondCanvas 25

Figure 2-12 FirstCanvas beneath opaque SecondCanvas 26

Figure 2-13 FirstCanvas beneath unmapped opaque SecondCanvas 26

Figure 2-14 Image of painted, transparent SecondCanvas on FirstCanvas 27

Figure 2-15 Transparent FirstCanvas beneath opaque SecondCanvas 28

Figure 2-16 Damage on unretained GrayCanvas after moving BlackCanvas 31

Figure 2-17 No damage on retained GrayCanvas after moving BlackCanvas 31

Figure 2-18 Unretained GrayCanvas damaged by unmapping BlackCanvas 32

Figure 2-19 GrayCanvas not damaged by unmapping SaveBehind BlackCanvas 33

Figure 2-20 Results of filling FirstCanvas after setting a canvas clipping path 34
Figure 2-21 Results of eoclipcanvas 35
Figure 2-22 BlackCanvas obscuring WhiteCanvas 38
Figure 2-23 WhiteCanvas made to obscure BlackCanvas 38
Figure 2-24 Canvas hierarchy 39
Figure 2-25 WhiteCanvas is now the child of SecondParent 40
Figure 2-26 New canvas hierarchy 40
Figure 2-27 A canvas and its overlay 43
Figure 2-28 A canvas and its erased overlay 44
Figure 2-29 StarCanvas 46
Figure 2-30 FileCanvas imaged onto SecondCanvas 48
Figure 2-31 StarCanvas imaged onto SecondCanvas 49
Figure 2-32 Image built with **buildimage** and imaged with **imagemaskcanvas** 51
Figure 2-33 Image with 0 bits painted black 51
Figure 2-34 The framebuffer's default cursor 52
Figure 2-35 MyCanvas with a crosshair cursor 54

Figure 4-1 The five steps in an event's distribution 81
Figure 4-2 Circles drawn in the canvas 89
Figure 4-3 Example canvas hierarchy 98
Figure 4-4 Example canvas hierarchy as it might appear on the screen 99
Figure 4-5 Example canvas hierarchy 101
Figure 4-6 The first pre-child interest matched 106
Figure 4-7 The second pre-child interest matched 106
Figure 4-8 The third pre-child interest matched 106
Figure 4-9 The first post-child interest matched 107
Figure 4-10 The second post-child interest matched 107
Figure 4-11 An example of drawing in the canvas 113
Figure 4-12 Mouse cursor over CanvasC 117
Figure 4-13 Moving the mouse from CanvasC to CanvasD 117
Figure 4-14 Moving the mouse from CanvasC to the framebuffer canvas 118

Figure 5-1 A simple class tree 133
Figure 5-2 A class tree with multiple inheritance 134
Figure 5-3 A single inheritance example 136
Figure 5-4 Dictionary stack before and during a **send** to MyScrollBar 142
Figure 5-5 Dictionary stack before and during a nested **send** 143
Figure 5-6 Class tree for **self** and **super** example 146
Figure 5-7 Basic class hierarchy for the multiple inheritance examples 166
Figure 5-8 Class hierarchy with a utility class 168
Figure 5-9 Class tree for LabeledDial example 170
Figure 5-10 A breadth-first order for LabeledDial's inheritance array 171
Figure 5-11 A depth-first order for LabeledDial's inheritance array 172

Figure 6-1 The lunar lander game 199

Preface

This manual provides a guide to programming in the NeWS® language. This language is supported as part of the X11/NeWS™ server, which itself forms a part of the OpenWindows™ environment.[1]

The NeWS interpreted programming language is based on the POSTSCRIPT® language.[2] Developed at Adobe Systems, the POSTSCRIPT language is a general programming language used primarily for specifying the visual appearance of printed documents. The NeWS language uses POSTSCRIPT language operators to display text and images on a graphics console. The NeWS language also provides operators and types that are extensions to the POSTSCRIPT language; many of these extensions handle the interactive aspects of window management that the POSTSCRIPT language does not consider.

This manual describes all the basic concepts of NeWS programming. The manual is a combination guide and reference to the NeWS language; the conceptual chapters are placed toward the front of the manual, and the reference chapters are placed toward the end. The conceptual chapters include code examples that demonstrate the use of NeWS operator and type extensions. The reference chapters provide descriptions of all the NeWS operators, types, and utilities. This manual assumes that the reader is familiar with the POSTSCRIPT language.

Summary of Contents

Chapter 1, "Introduction," provides an overview of NeWS programming.

Chapter 2, "Canvases," provides an introduction to NeWS canvases and canvas operations.

Chapter 3, "Processes," explains NeWS processes, process operations, and the server's scheduling policy.

Chapter 4, "Events," discusses NeWS events, event operations, and the server's event distribution mechanism.

Chapter 5, "Classes," describes the NeWS class mechanism, including all of the methods of the base class **Object**.

Chapter 6, "C Client Interface," describes the interface (known as CPS) that is provided for C clients.

[1] X Window System is a product of the Massachusetts Institute of Technology.

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

Chapter 7, "Debugging," describes the debugging facility provided with the server.

Chapter 8, "Memory Management," explains the server's reference counting and garbage collection facility and provides hints for debugging memory management problems.

Chapter 9, "NeWS Type Extensions," provides a reference to all the NeWS types, including descriptions of all the dictionary keys.

Chapter 10, "NeWS Operator Extentions," provides an alphabetical reference to all the NeWS operators.

Chapter 11, "Extensibility through NeWS Procedure Files," provides an alphabetical reference to all the NeWS utilities, as well as a description of the POSTSCRIPT language files that the server loads when it is initialized.

Appendix A, "NeWS Operators," contains an alphabetical list of all NeWS operators, including the syntax and a one-line description for each operator.

Appendix B, "Byte Stream Format," provides a reference to the server's byte stream format, including a description of all the server's token types.

Appendix C, "The Extended Input System," contains information about the Lite user interface; this user interface is still supported by the server, but is no longer being enhanced.

Appendix D, "Omissions and Implementation Limits," lists the operators that are provided by the POSTSCRIPT language but are not provided by the NeWS language, and it also summarizes the server's implementation limits.

For More Information

For information about the POSTSCRIPT language, see:

- *POSTSCRIPT Language Tutorial and Cookbook*[3]
- *POSTSCRIPT Language Reference Manual*[4]

For information about OpenWindows, see:

- *OpenWindows User's Guide*
- *DeskSet Environment Reference Guide*
- *OpenWindows Installation and Startup Guide*

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

- *X11/NeWS Server Guide*

For a summary of the changes to OpenWindows since the last release, see:

- *OpenWindows Release Notes*

---

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

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

Notational Conventions

This manual uses the following notational conventions:

- **`bold listing font`**

  This font indicates text or code typed at the keyboard during an interactive session with the operating system shell or with `psh`.

- `listing font`

  This font indicates information displayed by the computer during an interactive session. It it also used in code examples and textual passages to indicate use of the C programming language, and it is used for filenames, command names, and error names.

- sans serif font

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

- **bold font**

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

- *italic font*

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

- gray boxes

> Examples of interactive dialog with the operating system shell or with `psh` are shown in gray boxes.

- plain boxes

> Examples of POSTSCRIPT language code or C language code are shown in plain boxes.

1

Introduction

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

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

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

1.1. NeWS Programming: An Overview

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

The POSTSCRIPT Language

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

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

NeWS Types

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

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

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

Other NeWS type extensions are *opaque* and cannot be accessed as dictionaries. A full description of all NeWS type extensions is provided in Chapter 9, "NeWS Type Extensions."

NeWS Operators

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

A full description of all NeWS operator extensions is provided in Chapter 10, "NeWS Operator Extensions."

The X11/NeWS Server

The X11/NeWS server is a process that can exist on any graphics machine within a network, its function being to interpret and execute programs written in the NeWS language and to display the resulting graphics on the screen. The X11/NeWS server is neither a toolkit nor a user interface; it provides neither standards nor defaults for the creation and appearance of windows. The X11/NeWS server simply interprets and executes NeWS programs. User interfaces can thus be designed entirely by the programmer.

NeWS Processes

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

A full description of NeWS processes is provided in Chapter 3, "Processes."

Client-Server Communication

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

Typically, a client program contains two main sections. One section, which can be written in C, FORTRAN, or any other language, is used to perform the application's basic computations; this section is executed in the client process. The other section, which must be written in the NeWS language, is used to provide corresponding windows or graphics; this section is interpreted by the server process. The NeWS section of the client program can be detatched, sent to the server, and executed remotely with function calls. Sending code to the server in this

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

way provides a significant speed advantage when the client and server reside on different machines.

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

C Client Interface

Most programmers are likely to use C as the language of the client application. Therefore, the server provides a special interface facility that supports C client communication. The C client interface, named *CPS*, converts the client's NeWS code into functions callable by the client's C code. The C client interface is discussed in Chapter 6, "C Client Interface."

Programmers can also create their own interface facility for use with other languages. The server's byte stream format is discussed in Appendix B, "Byte Stream Format."

Canvases

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

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

Canvases exist in a hierarchy. The root of the server's canvas hierarchy is known as the *global root canvas.* (See "Multiple Screen Support," below, for the implications of the global root canvas.) Each canvas in the hierarchy can have any number of children; the display of each child canvas is clipped to the edges of its parent. Canvases overlap according to their positions in the hierarchy. When visible on the screen, opaque children obscure their parent. A canvas' children exist in an ordered list that determines their overlapping relationships. For a canvas to be visible on the screen, the canvas and all its ancestors must be mapped.

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

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

The NeWS language provides operator extensions for creating and manipulating canvases. A full account of canvases is provided in Chapter 2, "Canvases." The canvas dictionary keys are described in Chapter 9, "NeWS Type Extensions."

Imaging Model

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

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

Events

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

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

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

A full account of events is provided in Chapter 4, "Events." The event dictionary keys are described in Chapter 9, "NeWS Type Extensions."

Memory Management

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

The NeWS language provides the notion of *soft* references for programs that want to track objects without affecting the lifespan of the objects. A window manager

is an example of this type of program. A window manager has references to the canvases that it tracks, but the window manager does not want its references to prevent canvases from being garbage collected. In this type of situation, client programs should use soft references.

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

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

The NeWS language provides operators that aid in memory management. A full account of the memory management facilities is provided in Chapter 8, "Memory Management."

Color Support

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

Full information on all color-related types is provided in Chapter 9, "NeWS Type Extensions."

Font Support

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

For a description of the NeWS font dictionary structure, see Chapter 9, "NeWS Type Extensions."

Multiple Screen Support

You can run the server with more than one display screen attached to your machine. Each display screen has an associated canvas, known as a *framebuffer canvas* or *device canvas*, that covers the entire background of the screen. Each framebuffer canvas is a child of the server's global root canvas. The global root canvas and the framebuffer canvases are created when the server is initialized.

For information about programming with multiple screens, see Chapter 2, "Canvases." For information about installing multiple screens, see the *X11/NeWS Server Guide*.

1.2. POSTSCRIPT Language Files Used with the Server

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

This section describes some of the more important POSTSCRIPT language files. Full information on these files is provided in Chapter 11, "Extensibility through NeWS Procedure Files."

Classes

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

NeWS classes belong to a class hierarchy. The root of the hierarchy is class **Object**, which is implemented by the server. Other classes in the hierarchy can be provided by the client or by a toolkit.

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

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

Information on the class system is provided in Chapter 5, "Classes."

Debugging

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

Full information on using the debugger is provided in Chapter 7, "Debugging."

Utilities

The server provides many utilities that can be used in your NeWS code. These utilities are defined in POSTSCRIPT language files that are loaded when the server is initialized. For definitions of these utilities, see Chapter 11, "Extensibility through NeWS Procedure Files."

2

Canvases

A NeWS *canvas* is a region of the screen in which the client application can display text and graphics. Canvases are the basic drawing surfaces used to create objects such as windows and menus. Canvas boundaries are defined by POSTSCRIPT language paths and thus can be any shape; they need not be rectangular or even contiguous.

When visible on the screen, canvases can overlap. When this occurs, the hidden portion of a canvas can be stored offscreen and redisplayed when it is re-exposed. A canvas can be repositioned in the stack of overlapping canvases on the screen. A canvas can also be moved, reshaped, and resized.

Windows are usually composed of more than one canvas. For example, a window might use a separate canvas for each of the following items: a frame, scrollbars, a title bar, command buttons, and the drawing surface itself.

Each NeWS process can have a *current canvas*, which is the canvas that is manipulated by the drawing operations performed by that process. Any of the standard POSTSCRIPT language operators can be used to display text and graphics in NeWS canvases. In addition, the NeWS language provides operator extensions and utilities for manipulating canvases.

This chapter describes NeWS canvases and basic canvas operations.

The canvastype Extension

Each canvas is an object of type **canvastype**, which is a NeWS extension to the POSTSCRIPT language. Each **canvastype** object can be accessed as a POSTSCRIPT language dictionary. A canvas dictionary includes keys that describe the following properties (the keys are listed in parentheses):

- Ancestor and sibling relationships between canvases (**TopCanvas, BottomCanvas, CanvasAbove, CanvasBelow, TopChild, Parent**)
- The appearance of canvases on the screen (**Transparent, Mapped**)
- The handling of canvas storage (**Retained, SaveBehind**)
- How a canvas affects the distribution of events (**EventsConsumed, Interests**)
- The color properties of the canvas (**Color, Colormap, Visual, VisualList**)
- The cursor associated with the canvas (**Cursor**)

- Properties for keeping a canvas in shared memory (**SharedFile, RowBytes**)
- X11-related properties (**OverrideRedirect, BorderWidth, VisibilityInterest, SubstructureRedirect, XID**)
- The grabbed state of a canvas (**Grabbed, GrabToken**)

Many of these keys are discussed in this chapter; a full description of each key is provided in Chapter 9, "NeWS Type Extensions." See Chapter 4, "Events," for a description of how the **EventsConsumed** and **Interests** keys affect the distribution of events.

Canvas Operators

The NeWS language includes a variety of operator extensions to be used on canvases. The canvas operators provide the following functionality (the operator names are listed in parentheses):

- Creating canvas objects and overlays (**buildimage, createdevice, createoverlay, newcanvas**)
- Changing sibling relationships between canvases (**canvastobottom, canvastotop, insertcanvasabove, insertcanvasbelow**)
- Setting and getting a canvas' shape (**eoreshapecanvas, reshapecanvas, getcanvasshape**)
- Setting and getting canvas locations (**movecanvas, getcanvaslocation**)
- Setting and getting the current canvas (**setcanvas, currentcanvas**)
- Setting and getting a canvas' clipping path (**clipcanvas, eoclipcanvas, clipcanvaspath**)
- Writing a canvas to a file and reading the file back into a canvas (**eowritecanvas, eowritescreen, writecanvas, writescreen, readcanvas**)
- Imaging a canvas onto the current canvas (**imagecanvas, imagemaskcanvas**)
- Returning the global root canvas (**globalroot**)
- Returning the canvases under the current path or a specified point (**canvasesunderpath, canvasesunderpoint**)

Most of the canvas operators are described in this chapter. A list of all the NeWS operators is provided for quick reference in Appendix A, "NeWS Operators." A syntactic analysis and description of all NeWS operators is provided in Chapter 10, "NeWS Operator Extensions."

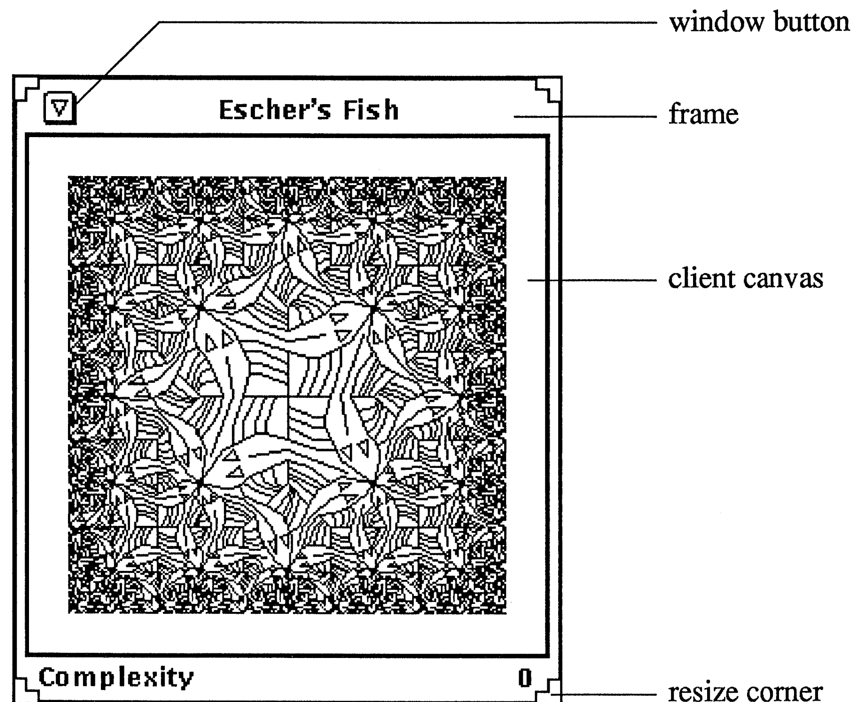
2.1. Basic Terms and Concepts

This section explains basic terms and concepts that you need to understand before you read the rest of this chapter.

Using Multiple Canvases to Create a Window

An application's window usually consists of more than one canvas. The following example shows a simple window that could be made of four canvases:

Figure 2-1 *A simple window*



The four canvases in this example are listed below:

- The client canvas
  A window's drawing surface is known as the *client canvas.*
- The frame canvas
  A window commonly has a *frame canvas* that manages the user interface for the window. The client canvas sits on top of the frame canvas.
- The window button canvas
  A window's frame often has command buttons that sit on top of it. The command buttons implement user interaction. In this example, a window button is provided; the user can click the mouse over the button to close the application's window to an icon.
- The resize corners canvas
  A window's frame often provides resize corners. The user can drag the mouse inward or outward from one of these corners to resize the window. In this example, the four resize corners make up a single canvas that sits on top of the frame canvas; each corner is one part of this noncontiguous canvas. (You can make a noncontiguous canvas by assigning any number of closed paths, four in this example, to be the canvas' shape.)

The canvases that compose an application's window are part of a canvas hierarchy, as is explained in the next section.

The Canvas Hierarchy

The server maintains a *canvas hierarchy.* The root of the canvas hierarchy is the *global root canvas.* Each canvas in the hierarchy can have any number of *children.* The global root canvas has one child canvas, known as the *framebuffer canvas*, for each display screen (framebuffer) that is attached to the machine on which the server is running. A screen's framebuffer canvas, sometimes called the screen's *root canvas* or *device canvas*, covers the entire background of the screen. The global root canvas and the framebuffer canvases are created when the server is initialized.

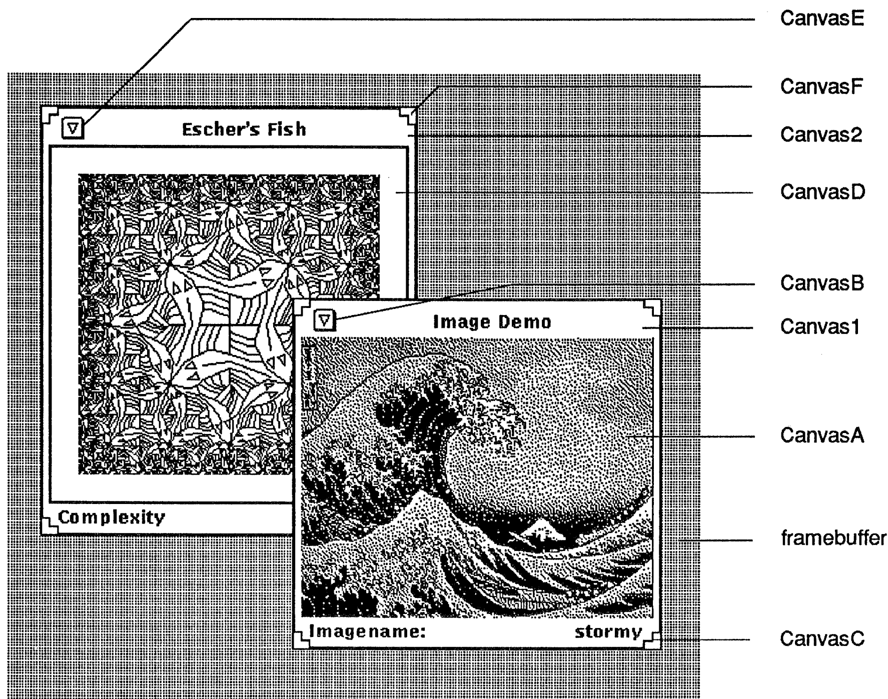
Clients can create children of the framebuffer canvases, and each such canvas can have its own children and grandchildren. The canvases form a hierarchy much like a family tree, except that each child canvas has just one parent.

Most of the time, only one display screen is used. Therefore, the rest of this chapter discusses the case in which only one framebuffer canvas exists. For more information about programming with multiple screens, see Section 2.10, "Using Multiple Screens." For more information about installing multiple screens, see the *X11/NeWS Server Guide.*

Canvases overlap according to their positions in the canvas hierarchy. Each child canvas sits on top of its parent. The display of each child canvas is clipped to the edges of its parent. If part or all of a child canvas is moved off its parent canvas, the part of the child canvas that extends beyond its parent's edges is not visible on the screen.

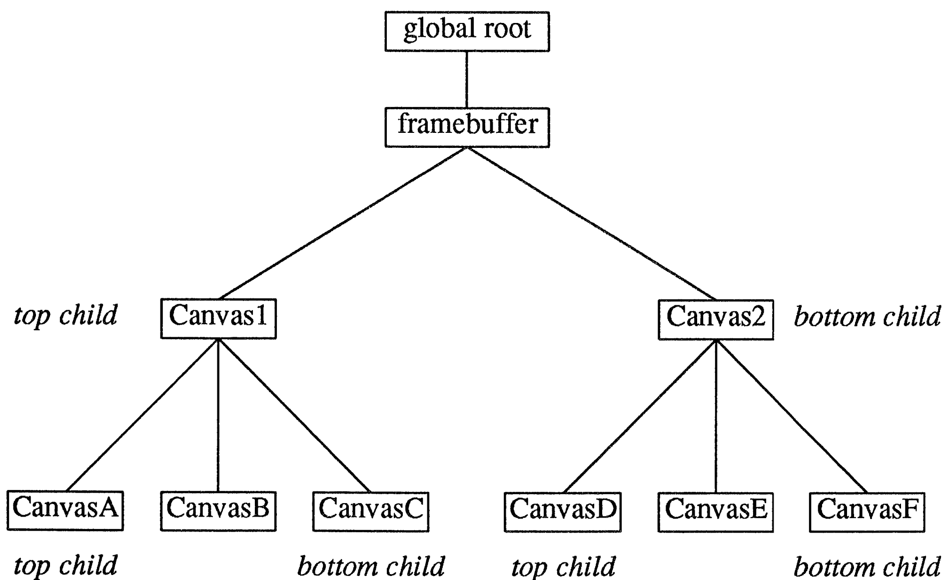
The following figure illustrates two overlapping windows on the screen, and it shows one way these two windows could be subdivided into canvases:

Figure 2-2 *Two overlapping windows*



The example hierarchy associated with these overlapping canvases is shown in the following figure:

Figure 2-3 *Example hierarchy*



In this example, Canvas1 and Canvas2 are children of the framebuffer canvas. Canvas1 and Canvas2 each have three children.

When a parent canvas has more than one child, its children are known as *siblings* of each other. Sibling canvases exist in an ordered list that determines their overlapping relationships. When two siblings overlap, the sibling that is closer to the head of its parent's sibling list overlaps the sibling that is closer to the tail of the sibling list. The sibling that is at the head of the list is known as its parent's *top child*, and the sibling that is at the tail of the list is known as the *bottom child*. The top child is on top of its parent's stack of sibling canvases on the screen, and the bottom child is on the bottom of the stack. The terms *front* and *back* are sometimes used for top and bottom.

In this example, Canvas1 and Canvas2 are siblings. Canvas1 is the top child of the framebuffer canvas; Canvas2 is the bottom child of the framebuffer canvas. Thus, Canvas1 obscures Canvas2 where they overlap.

The children of Canvas1 are ordered in their own sibling list, as are the children of Canvas2. However, these siblings do not overlap. If the children of Canvas1 overlapped each other, CanvasA would obscure CanvasB, and CanvasB would obscure CanvasC. If the children of Canvas2 overlapped each other, CanvasD would obscure CanvasE, and CanvasE would obscure CanvasF. (The above illustration of the canvas hierarchy orders siblings left to right from the head of the list to the tail of the list.)

By default, a newly created child canvas becomes the top child of its parent. The canvas hierarchy can be changed with various operators and with some of the canvas dictionary keys. A canvas can be inserted into a different position in its sibling list, or a canvas can be given a different parent.

A canvas' *descendants* (the canvas' children, grandchildren, etc.) are located on the hierarchy branches that emanate leafward from the canvas. A canvas' *ancestors* (the canvas' parent, grandparent, etc.) are located on the hierarchy branch that emanates rootward from the canvas. In the previous example, the descendants of Canvas1 are CanvasA, CanvasB, and CanvasC. The ancestors of CanvasA are Canvas1, the framebuffer canvas, and the global root canvas. If CanvasA had a child, that child's ancestors would be CanvasA, Canvas1, the framebuffer canvas, and the global root canvas.

Opaque and Transparent Canvases

A canvas is either *opaque* or *transparent*, depending on the boolean value of its **Transparent** key. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. If drawing operations are performed on a transparent canvas, the drawn images appear on the canvas(es) immediately beneath the transparent canvas (that is, on its parent or on siblings that are immediately beneath it). Although a transparent canvas does not have its own drawing surface, it can define screen areas that are sensitive to input just as any opaque canvas can. See Chapter 4, "Events," for an explanation of events, interests, and input handling.

By default, children of the framebuffer canvas are opaque and all other canvases in the hierarchy are transparent. You can change a canvas' opaque/transparent status by changing the value of its **Transparent** key.

Visibility and Mapping

A canvas is *visible* if an image can be seen on the screen when drawing operations are performed on the canvas. This definition applies to transparent canvases as well as opaque canvases; if a transparent canvas is visible, drawing operations on the transparent canvas result in an image on the canvas(es) beneath it.

A canvas is either *mapped* or *unmapped*, depending on the boolean value of its **Mapped** key. A canvas must be mapped before its contents can be visible on the screen. In fact, all of the following conditions must be fulfilled before a canvas is visible:

- The canvas and all of its ancestors must be mapped (the value of their **Mapped** keys must be **true**).
- The canvas must not be clipped away by its parent. The portions of a canvas that fall outside the boundary of its parent are not visible.
- The canvas must not be overlapped by an opaque canvas. The portions of a canvas that are overlapped by opaque canvases are not visible.

If no drawing operations have been performed on a canvas that meets the above three criteria, the empty canvas might not be noticed on the screen. Such a canvas is still considered to be visible because drawing operations can be performed at any time to render images to the screen and because the canvas visibly affects the display of other canvases beneath it.

Canvas Damage

The server considers a canvas to be *damaged* if all or part of its image is incorrect and needs to be redrawn. For example, a canvas may be damaged when a canvas by which it was previously obscured is moved away; the damaged region is the newly exposed area.

The first time a canvas is damaged since its last repair, the server informs interested processes of the damage by sending them a *damage event*. Client applications should be prepared to repair canvas damage on any of their canvases. See Section 2.4, "Canvas Damage: When to Expect It, How to Fix It, How to Avoid It," for a complete description of canvas damage. See Chapter 4, "Events," for a description of how to express interest in events and for an explanation and example of damage events.

Retained and Unretained Canvases

A canvas is either *retained* or *unretained*, depending on the boolean value of its **Retained** key. Any portion of a retained canvas that is not visible (because it is obscured, clipped, or unmapped) is saved offscreen. When an invisible area of a retained canvas is exposed, the offscreen copy is simply moved onto the screen, eliminating the need to redraw the newly exposed area. Thus, retained canvases can be used to reduce canvas damage.

Retained canvases usually perform much better than unretained canvases with window management operations such as moving and mapping canvases. However, retained canvases can be extremely costly in terms of memory, especially on color displays. Also, a slight performance penalty is associated with painting on a retained canvas.

Setting a canvas to be retained is just a hint to the server; the server may choose to ignore the hint. Clients should not depend on the server saving a copy of a retained canvas' invisible areas.

A transparent canvas does not have its own retained image. Instead, a transparent canvas shares the retained image of its parent. Changing the retained status of a transparent canvas has no effect on either the transparent canvas or its parent.

Rooted and Unrooted Canvases

A *rooted* canvas is part of the canvas hierarchy; an *unrooted* canvas is not. An unrooted canvas can never be mapped, but its image can be painted onto the current canvas. See Section 2.8, "Canvases, Files, and Imaging Procedures," for more information about unrooted canvases and imaging.

Coordinate Systems

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

In the NeWS language, each canvas represents a separate "user space" with its own coordinate system, and the device space corresponds to the screen rather than to a printer. A current transformation matrix is still used to store the current transformation between the user and device coordinate systems, but in NeWS, each process has its own CTM as a part of its graphics state. A process' current coordinate system is given by its CTM.

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

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

When the first NeWS process is created, its current coordinate system is initialized to the default coordinate system of the framebuffer canvas. A child process inherits the current coordinate system of its parent.

The default coordinate system of the framebuffer canvas is initialized so that the origin is in the screen's lower-left corner, the positive y axis extends vertically upward, and the positive x axis extends horizontally to the right. In the current implementation, the framebuffer canvas' default coordinate system is initialized so that the length of a unit in the y coordinate direction corresponds to the vertical pixel dimension, and the length of a unit in the x coordinate direction

corresponds to the horizontal pixel dimension; in a future implementation, the framebuffer canvas' default coordinate system might be initialized so that a unit in either coordinate direction corresponds to exactly 1/72 of an inch, consistent with the standard POSTSCRIPT language.

NOTE *The default coordinate system of the screen (the device coordinate system) typically has its origin in the upper-left corner, the positive y axis extending vertically downward, the positive x axis extending horizontally to the right, and units of one pixel in both coordinate directions. Because the default coordinate system of the screen typically has its origin in the upper-left corner, whereas the default coordinate system of the framebuffer canvas has its origin in the lower-left corner, the initial CTM simply provides the appropriate transformation between these two coordinate systems. You can use the POSTSCRIPT language operator* **currentmatrix** *to inspect the CTM.*

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

2.2. Basic Canvas Operations

This section describes basic canvas operations such as creating, mapping, shaping, and moving canvases. An example is given for each operation, but the examples in this section are cumulative; be sure to try these examples in the given order. You can start an interactive `psh` session and type each short example sequentially. You start an interactive `psh` session by typing the word **psh** followed by a [Return] and then typing the word **executive** followed by a [Return] (see the `psh` manual page in the *X11/NeWS Server Guide* for more information).

Creating Canvases

When the server is initialized, the **createdevice** operator is called to create a canvas that covers the entire background of the initial display screen. The initial display screen is given by the value of the `FRAMEBUFFER` environment variable, which defaults to `/dev/fb`. You can create background canvases for additional screens with the **createdevicecanvas** utility (see Section 2.10, "Using Multiple Screens"). A screen's background canvas is known as its *device canvas* or *framebuffer canvas*. The framebuffer canvas associated with the screen that currently contains the mouse pointer is known as the *current framebuffer canvas*. The **framebuffer** variable in **systemdict** can be used to refer to the current framebuffer canvas.

When a client program makes a connection to the server, a copy of the **framebuffer** value from **systemdict** is placed in that NeWS process' **userdict**. Any canvas that you wish to create immediately on top of this background must have that framebuffer canvas specified as its parent.

The following operator creates a canvas with a specified parent:

pcanvas **newcanvas** ncanvas

This operator creates a new canvas, *ncanvas*, whose parent is *pcanvas*. If a framebuffer canvas is used as the *pcanvas* argument, the new canvas is opaque by default. If the parent is not the framebuffer, the new canvas is transparent by default.

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

```
myprompt% psh
executive                                   % Start an interactive psh session.
Welcome to X11/NeWS Version 2.1

/FirstCanvas framebuffer newcanvas def       % Create a new canvas.
```

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

A newly created canvas such as FirstCanvas is not immediately ready for use. First, you must set the canvas' shape and default coordinate system. Before you can use standard drawing operators, you must make the canvas be the current canvas. And before you can see an image on the screen, you must map the canvas to the screen. These steps are described and demonstrated in the following three sections.

Setting a Canvas' Shape and Coordinate System

After you create a canvas, you must give it a shape and default coordinate system. The following operator accomplishes these two tasks:

canvas **reshapecanvas** –

This operator sets the shape of *canvas* to be the same as the current path. It also sets *canvas*' default transformation matrix so that *canvas*' default coordinate system is the same as the current coordinate system. If *canvas* is the current canvas, **reshapecanvas** sets *canvas*' default transformation matrix so that the same default coordinate system is maintained after *canvas* changes shape; it also performs an implicit **initmatrix** and sets the current clipping path (in the graphics state) to be the same as *canvas*' new shape. If you reshape a parent canvas, each child canvas maintains the same distance from the upper-left corner of the bounding box of the parent.

The following example uses **reshapecanvas** to establish a shape and default coordinate system for the canvas defined in the previous example. This example uses the **rectpath** utility to create a path to which the canvas can be shaped. The **rectpath** utility is provided by the POSTSCRIPT language extensibility files associated with the server; **rectpath** adds a rectangle to the current path, given the *x* and *y* coordinates of the rectangle's origin, the rectangle's *width*, and the rectangle's *height*. See Chapter 11, "Extensibility through NeWS Procedure Files," for a complete definition of **rectpath**.

```
0 0 250 250 rectpath              % Create a path to which the
                                  % new canvas can be shaped.

FirstCanvas reshapecanvas         % Reshape the canvas to the
                                  % current path, and set the
                                  % canvas' default coordinate
                                  % system to be the same as the
                                  % current coordinate system.
```

When a psh process is first started, its CTM is initialized to the default coordinate system of the framebuffer canvas. This example did not change the CTM before calling **reshapecanvas**. Therefore, the default coordinate system of FirstCanvas is set to be the same as the CTM, which is still the default coordinate system of the framebuffer canvas. Thus, the default coordinate system of FirstCanvas has its origin in the lower-left corner of the screen with the positive x axis to the right and the positive y axis up. The canvas is shaped so that its lower-left corner is at the origin of its coordinate system. The lower-left corner of a canvas is often chosen to be its origin; however, you can use **reshapecanvas** to place a canvas anywhere with respect to the origin of its default coordinate system.

The NeWS language also provides an operator named **eoreshapecanvas**. This operator is identical to **reshapecanvas** except that it uses the even-odd rule, rather than the non-zero winding number rule, to interpret the path argument. For information on these rules, see the *POSTSCRIPT Language Reference Manual*. For a description of **eoreshapecanvas**, see Chapter 10, "NeWS Operator Extensions."

Mapping Canvases to the Screen

No operator exists for mapping canvases to the screen; instead, you map canvases by setting the **Mapped** key of the **canvastype** dictionary to **true**. When you map a canvas, it becomes visible on the screen within the borders of its parent, provided that the following conditions are fulfilled:

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

To retrieve and establish values for any read/write NeWS dictionary key, you can use the POSTSCRIPT language operators **get** and **put** respectively. The following example uses **get** to inspect the value of FirstCanvas' **Mapped** key, and then it uses **put** to set the values of FirstCanvas' **Retained** and **Mapped** keys:

```
FirstCanvas /Mapped get ==
false

FirstCanvas /Retained false put
FirstCanvas /Mapped true put
```

Notice that you cannot see FirstCanvas on the screen even after it is mapped; a mapped canvas might not be noticeable on the screen if you have not drawn on it. Before you draw on FirstCanvas, you need to make FirstCanvas be the current canvas.

Setting the Current Canvas

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

canvas **setcanvas** –

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

The following example sets the current canvas to be FirstCanvas:

```
FirstCanvas setcanvas
```

Drawing on Canvases

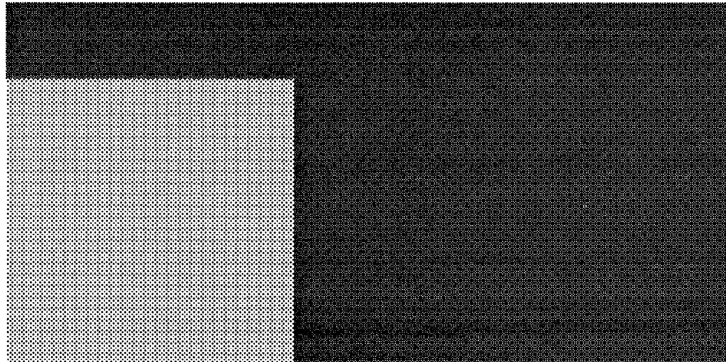
Once a canvas is the current canvas, you can perform drawing operations on it. For example, you can use the **fillcanvas** utility to fill a canvas with a color. The **fillcanvas** utility is included in the POSTSCRIPT language extensibility files provided with the server. The utility takes a single argument, which can be an integer or a color; **fillcanvas** paints the canvas and sets the current color to be the specified value. See Chapter 11, "Extensibility through NeWS Procedure Files," for a complete description of **fillcanvas**.

The following example fills FirstCanvas with gray:

```
0.88 fillcanvas
```

When FirstCanvas is painted gray, it appears at the bottom-left corner of the framebuffer canvas because that is where it was previously positioned with the **reshapecanvas** operator. FirstCanvas' appearance is illustrated in the following figure:

Figure 2-4 *FirstCanvas mapped and filled with gray*

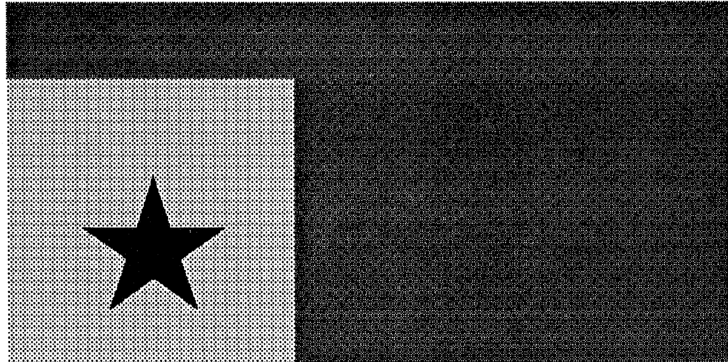


The example below draws a black star on FirstCanvas.

```
/starpath {                         % x, y -> starpath
    translate                       % Move origin to x,y.
    0 0 moveto                      % Move currentpoint to origin.
    4 {
        125 0 translate             % Go forward 125.
        0 0 lineto                  % Draw line.
        -144 rotate                 % Turn right 144 degrees.
    } repeat
    closepath                       % Draw last line.
} def

/paintstar {                        % color, x, y -> starimage
    gsave                           % Paints an image of a star
    starpath setgray fill           % with the specified color
    grestore                        % at location x, y.
} def

0 65 120 paintstar                  % Paint a star on FirstCanvas.
```

The following figure illustrates FirstCanvas with the star drawn on it:

Figure 2-5 *FirstCanvas with star*

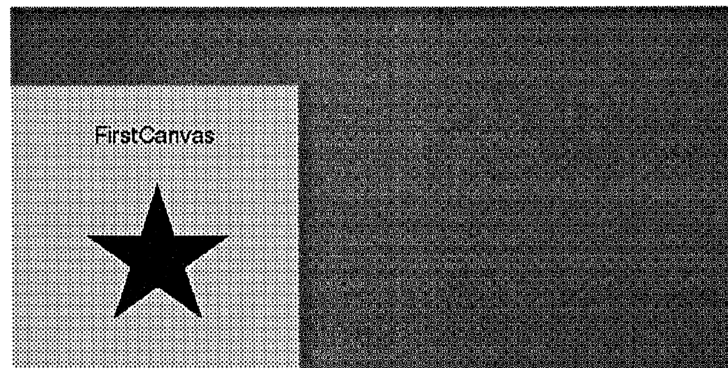


You can also draw text strings on canvases, using any of the POSTSCRIPT language operators or the NeWS extensions. The following example writes a string on FirstCanvas. This example uses the **cshow** utility, which is provided by the POSTSCRIPT language extensibility files. The **cshow** utility centers and prints a text string at the current point. See Chapter 11, "Extensibility through NeWS Procedure Files," for a complete description of **cshow**.

```
0 setgray
/Helvetica findfont 20 scalefont setfont
125 200 moveto
(FirstCanvas) cshow
```

The following figure illustrates the text string written on FirstCanvas:

Figure 2-6 *FirstCanvas with text string*



Moving Canvases

You can move a canvas to any location. However, the display of a canvas is clipped to its parent's boundaries. If a canvas is moved or reshaped so that parts of the canvas fall outside of its parent's boundaries, those parts of the canvas do not appear on the screen when the canvas is mapped.

The following operator moves a canvas to the specified location:

x y movecanvas –
x y canvas movecanvas –

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

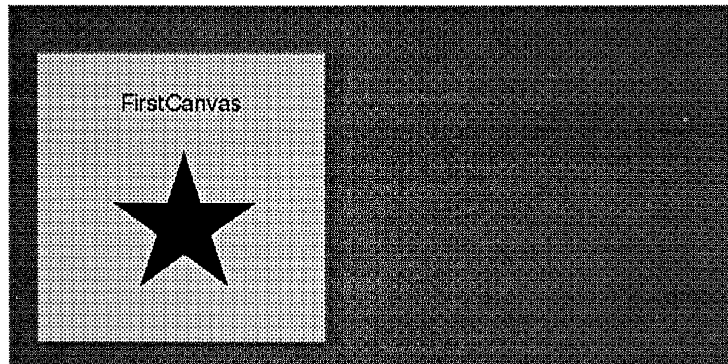
If the *canvas* argument is specified, the operator moves that canvas so that the origin of its default coordinate system is at the coordinates *x* and *y* in the current coordinate system.

In the following example, FirstCanvas is moved so that its origin is at (25, 25) in the framebuffer canvas' coordinate system:

```
25 25 movecanvas
```

The appearance of FirstCanvas is now as follows:

Figure 2-7 *FirstCanvas moved to 25, 25 in framebuffer canvas' coordinate system*



The same result could have been obtained with the following code:

```
framebuffer setcanvas
25 25 FirstCanvas movecanvas
```

Note that the **movecanvas** operator moves both the canvas and its default coordinate system. In this example, the origin of FirstCanvas' default coordinate system remains at FirstCanvas' lower-left corner, but FirstCanvas and its default coordinate system are now offset from the framebuffer canvas' default coordinate system.

Getting the Location of a Canvas

The following operator returns the coordinates of a canvas' origin:

canvas **getcanvaslocation** x y

This operator returns two integers, which specify the *x* and *y* location of the origin of *canvas*' default coordinate system. This location is given relative to the origin of the current coordinate system (rather than the origin of the parent canvas).

FirstCanvas was moved in the previous example. The following example returns the new coordinates of FirstCanvas, relative to the framebuffer canvas' default coordinate system:

```
framebuffer setcanvas              % Set the current coordinate
                                   % system to be the framebuffer's
                                   % default coordinate system.
FirstCanvas getcanvaslocation      % Return the location of FirstCanvas
pstack                             % relative to the current
25 25                              % coordinate system.
clear
```

Destroying Canvases

The NeWS language does not provide an operator for destroying a canvas; even when unmapped, a canvas continues to exist. A canvas is destroyed only when the last reference to the canvas is removed (see Chapter 8, "Memory Management"). If a canvas is still mapped when the last reference to it disappears, the canvas' image is removed from the screen as part of the garbage collection process.

In this section's examples, FirstCanvas has only one reference to it: the name FirstCanvas. This reference can be removed with the **undef** operator, as follows:

```
FirstCanvas /Mapped false put
userdict /FirstCanvas undef

quit                               % Quit psh.
```

In the above example, FirstCanvas would have been destroyed even without **undef** when `psh` was exited. (When the `psh` connection is broken by the **quit** operator, all objects defined in the `psh` process' **userdict** are undefined and the associated memory is reclaimed.)

2.3. Using the Transparent and Opaque Properties of Canvases

By default, children of the framebuffer canvas are opaque and all other canvases in the hierarchy are transparent. This section uses examples to illustrate the following aspects of transparent and opaque canvases:

- An opaque canvas visually hides all canvases underneath it; a transparent canvas does not.

- Anything painted on a transparent canvas is actually painted on the canvas(es) immediately beneath it.
- Making a parent canvas transparent does not affect the opaque status of its opaque children.

A transparent canvas is especially useful for defining an area that is sensitive to input but that has no drawing surface of its own. Input handling is described in Chapter 4, "Events."

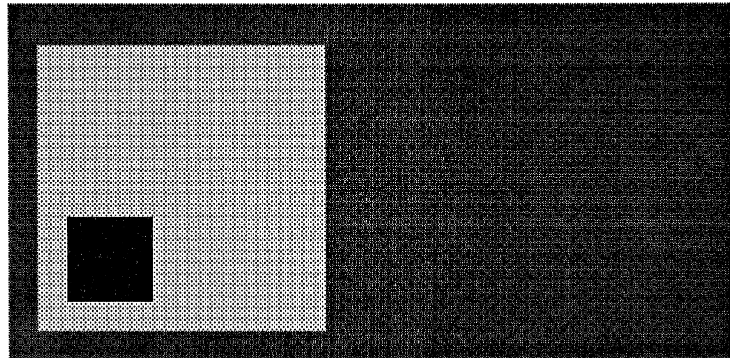
Opaqueness and Transparency

The example in this section demontrates that opaque canvases obscure other canvases, but transparent canvases do not. The example uses two canvases: FirstCanvas and SecondCanvas. FirstCanvas is a child of the framebuffer canvas, and SecondCanvas is a child of FirstCanvas. By default, FirstCanvas is opaque and SecondCanvas is transparent. To make SecondCanvas opaque, its **Transparent** key is set to **false**. FirstCanvas is painted gray; SecondCanvas is painted black.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/FirstCanvas framebuffer              % Create FirstCanvas.
newcanvas def
0 0 250 250 rectpath                  % Create a path.
FirstCanvas reshapecanvas             % Shape FirstCanvas to path.
FirstCanvas /Retained true put        % Make FirstCanvas retained.
FirstCanvas setcanvas
0.88 fillcanvas                       % Paint FirstCanvas.
FirstCanvas /Mapped true put          % Map FirstCanvas.
25 25 movecanvas                      % Move FirstCanvas.

/SecondCanvas FirstCanvas             % Create SecondCanvas.
newcanvas def
0 0 75 75 rectpath
SecondCanvas reshapecanvas            % Reshape SecondCanvas.
SecondCanvas /Transparent get         % Prove the canvas is transparent
pstack                                % by default.
true

clear

SecondCanvas /Transparent false put   % Make SecondCanvas opaque.
SecondCanvas /Mapped true put         % Map SecondCanvas.
SecondCanvas setcanvas                % Make it the current canvas.
25 25 movecanvas
0 fillcanvas                          % Paint SecondCanvas.
```

The following figure illustrates SecondCanvas and FirstCanvas:

Figure 2-8 *FirstCanvas and its child, SecondCanvas*

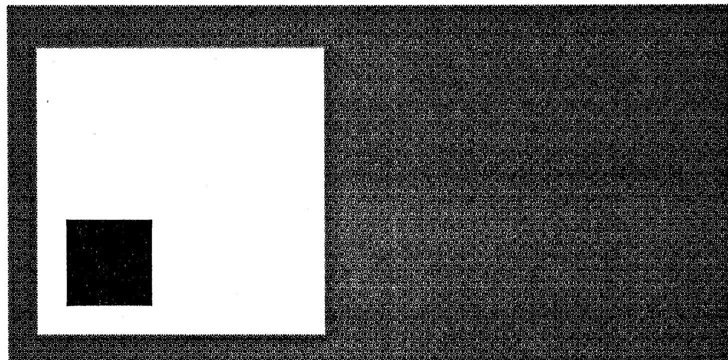


Next, FirstCanvas is painted white. Because SecondCanvas is opaque, it obscures FirstCanvas beneath it.

```
FirstCanvas setcanvas
1 fillcanvas
```

The new appearance of FirstCanvas is illustrated below:

Figure 2-9 *Newly painted FirstCanvas beneath opaque SecondCanvas*

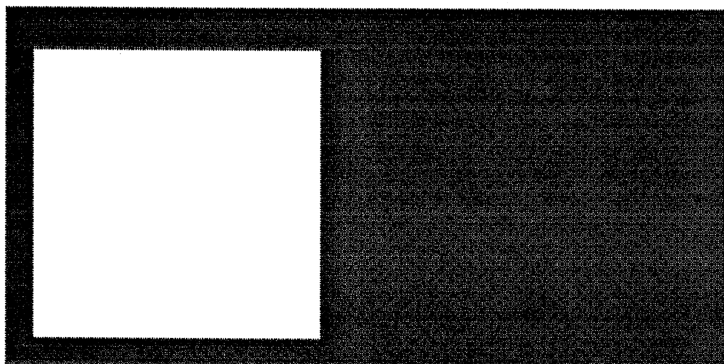


Now, SecondCanvas is made transparent, allowing FirstCanvas to show through:

```
SecondCanvas /Transparent true put
```

The following figure illustrates the disappearance of SecondCanvas when it is made transparent:

Figure 2-10 *FirstCanvas beneath transparent SecondCanvas*

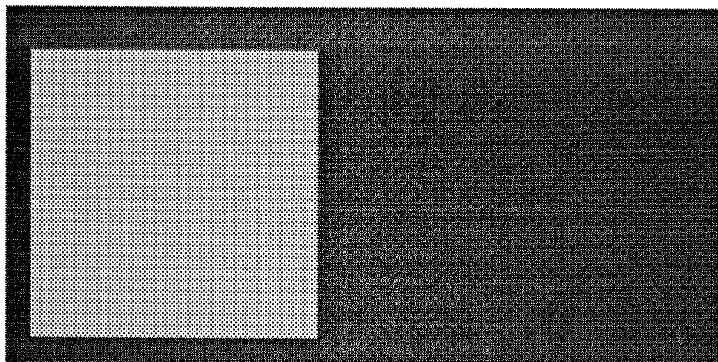


If FirstCanvas is painted while SecondCanvas is transparent, SecondCanvas does not obscure the new painting in FirstCanvas:

```
0.88 fillcanvas
```

The following figure illustrates the newly painted FirstCanvas:

Figure 2-11 *Newly painted FirstCanvas beneath transparent SecondCanvas*



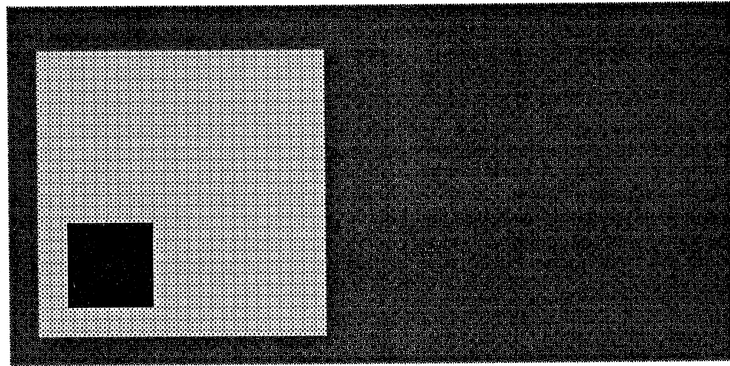
Painting on a Transparent Canvas

This subsection continues the previous example to illustrate how painting on a transparent canvas differs from painting on an opaque canvas. First, SecondCanvas is made opaque again and painted black. (When a canvas is changed from transparent to opaque, it automatically becomes unretained and receives damage. Therefore, SecondCanvas must be repainted regardless of its original retained status. See Section 2.4, "Canvas Damage: When to Expect It, How to Fix It, How to Avoid It," for a complete discussion of damage.)

```
SecondCanvas /Transparent false put
SecondCanvas setcanvas 0 fillcanvas
```

FirstCanvas and SecondCanvas now have the following appearance:

Figure 2-12 *FirstCanvas beneath opaque SecondCanvas*

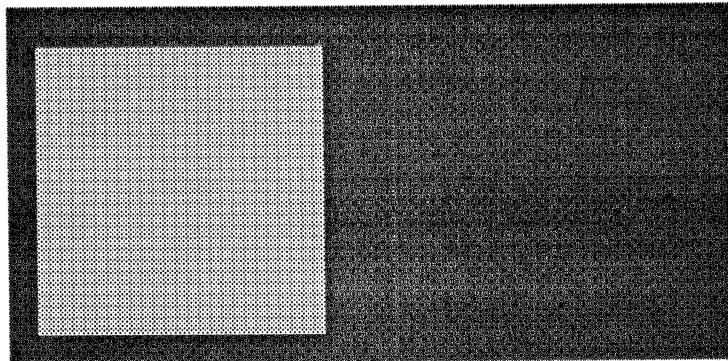


When the opaque SecondCanvas is unmapped, its image disappears.

```
SecondCanvas /Mapped false put
```

The disappearance of SecondCanvas is shown below:

Figure 2-13 *FirstCanvas beneath unmapped opaque SecondCanvas*

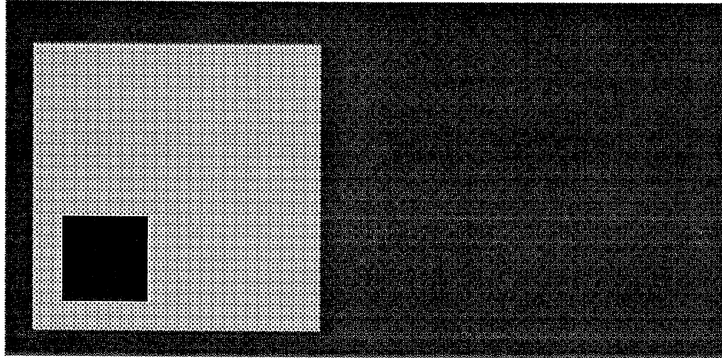


Next, SecondCanvas is made transparent, is mapped, and is painted black. When SecondCanvas is painted black, its black image is actually painted on FirstCanvas beneath it.

```
SecondCanvas /Transparent true put
SecondCanvas /Mapped true put
0 fillcanvas
```

SecondCanvas and FirstCanvas now appear as follows:

Figure 2-14 *Image of painted, transparent SecondCanvas on FirstCanvas*



If the transparent SecondCanvas is now unmapped, its black image remains on the screen because it was painted onto FirstCanvas. To prove that its image has become a part of FirstCanvas, you can unmap and map FirstCanvas and the black image remains.

```
SecondCanvas /Mapped false put         % Unmap SecondCanvas.
                                       % Its image remains.

FirstCanvas /Mapped false put          % Unmap FirstCanvas.
                                       % Its image disappears.

FirstCanvas /Mapped true put           % Map FirstCanvas.
                                       % Its image returns, complete
                                       % with the black square that
                                       % was painted on it previously
                                       % by SecondCanvas.
```

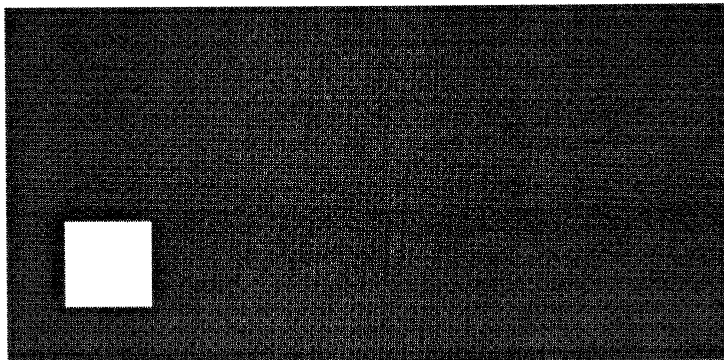
Making a Parent Canvas Transparent

When a parent canvas is made transparent, its opaque children are not affected and remain opaque. This behavior is demonstrated by the following code (a continuation of the previous example):

```
FirstCanvas setcanvas .88 fillcanvas     % Paint FirstCanvas gray.
SecondCanvas /Transparent false put      % Make SecondCanvas opaque.
SecondCanvas /Mapped true put            % Map SecondCanvas.
SecondCanvas setcanvas 1 fillcanvas      % Paint SecondCanvas white.

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

The following figure illustrates the disappearance of transparent FirstCanvas beneath opaque SecondCanvas:

Figure 2-15 *Transparent FirstCanvas beneath opaque SecondCanvas*



If you drew on FirstCanvas now, the images would be painted on the frame-buffer canvas because FirstCanvas is transparent.

The following code makes the parent opaque again and restores its previous appearance:

```
FirstCanvas /Transparent false put
FirstCanvas setcanvas 0.88 fillcanvas

quit                                    % Quit psh.
```

2.4. Canvas Damage: When to Expect It, How to Fix It, How to Avoid It

This section lists the situations in which damage can occur, explains the damage repair procedure, and discusses how to reduce damage by using retained canvases and **SaveBehind** canvases. Note that canvas damage cannot be eliminated altogether; clients must be prepared for damage on any of their canvases.

When is a Canvas Damaged?

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

- An unretained canvas is damaged when a canvas by which it was previously obscured is unmapped or moved away; only the newly exposed parts of the unretained canvas are damaged.
- The visible parts of an unretained canvas are damaged when the canvas is mapped to the screen.
- The invisible parts of an unretained canvas are damaged when its **Retained** key is changed from **false** to **true**. Also, a canvas that is retained by default is damaged when it is created (for details, see the *NOTE* in the subsection "Avoiding Canvas Damage with Retained Canvases").
- A mapped canvas, either retained or unretained, is damaged when it is reshaped.
- An unmapped, retained canvas is damaged when it is reshaped.

- When a transparent canvas is made opaque, it becomes unretained and the visible parts of the canvas are damaged.
- When a canvas is made transparent, the unretained canvases beneath it are damaged.
- Damage occurs to an unretained canvas when invisible portions of the canvas are copied onto visible portions of the canvas with the **copyarea** operator; damage only occurs to the visible portions of the copied area's destination that are to receive the image of previously invisible parts of the canvas. See Chapter 10, "NeWS Operator Extensions," for a description of **copyarea**.

A transparent canvas never receives damage. Instead, damage may be received by the canvas(es) beneath the transparent canvas.

Repairing Canvas Damage

When a canvas is initially damaged, the server sends a damage event to processes interested in damage on that canvas; a damage event has **/Damaged** in its **Name** field and a copy of the affected canvas in its **Canvas** field. After receiving a damage event, the client program should repair the damage by drawing the damaged parts of the canvas. The client can determine which parts of a canvas are damaged by executing the **damagepath** operator; **damagepath** returns a path that outlines the damaged regions (see the description of the **damagepath** operator in Chapter 10, "NeWS Operator Extensions").

If the client does not immediately repair the canvas and damage continues to occur, the server sends no additional damage events to the client. Instead, the server updates the record of the canvas' damage by adding the outline of the newly damaged region to the path returned by **damagepath**. Eventually, the client should request a copy of this record and repair all the damage. For an example of a damage event and subsequent repair, see the subsection "Damage Events" in Section 4.6, "System-Generated Events," of Chapter 4, "Events."

Avoiding Canvas Damage with Retained Canvases

One strategy for avoiding damage on a canvas is to make the canvas retained. When an invisible portion of a retained canvas is exposed, the canvas does not usually receive damage. On monochrome screens, retained canvases usually perform much better than unretained canvases when they are mapped or moved. On color screens, retained canvases usually consume too much memory to be useful.

Retaining an image offscreen cannot eliminate damage in all situations. For example, a retained canvas is damaged when it is reshaped. Also, the **Retained** key is just a performance hint that may be ignored. Clients should always be prepared for canvas damage, even on retained canvases.

Each system has a *retain threshold* that specifies the number of bits per pixel below which a canvas has its **Retained** key automatically set to **true** when the canvas is created. However, if your application desires that a canvas be retained, you should always set the **Retained** key explicitly. On all screens, the framebuffer canvas is unretained by default.

The default retain threshold is one bit per pixel, meaning that canvases on monochrome screens are retained by default and canvases on color screens are unretained by default. You may set the default retain threshold with the

setretainthreshold operator. For example, `8 setretainthreshold` would cause new canvases on a color screen to be retained.The server handles damage events for the framebuffer canvas, repainting the background as necessary.

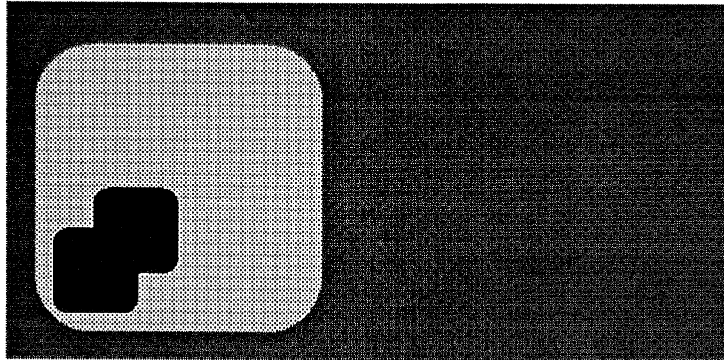
A transparent canvas does not have its own retained image. Instead, a transparent canvas shares the retained image of its parent. Changing the retained status of a transparent canvas has no effect on either the transparent canvas or its parent.

The following example demonstrates the damage that occurs to an opaque unretained canvas when another opaque canvas is moved across it. This example uses the **rrectpath** utility, which is similar to **rectpath** except that the rectangular path is given rounded corners (see Chapter 11, "Extensibility through NeWS Procedure Files," for more information).

```
myprompt% psh
executive
Welcome to X11/NeWS ,Version 2.1

/GrayCanvas framebuffer                    % Create GrayCanvas
newcanvas def
25 25 translate
50 0 0 250 250 rrectpath
GrayCanvas reshapecanvas
GrayCanvas /Retained false put
GrayCanvas /Mapped true put
GrayCanvas setcanvas 0.88 fillcanvas

/BlackCanvas GrayCanvas                    % Create BlackCanvas
newcanvas def
15 15 translate
newpath 15 0 0 75 75 rrectpath
BlackCanvas reshapecanvas
BlackCanvas /Transparent false put
BlackCanvas /Mapped true put
BlackCanvas setcanvas
0 fillcanvas

50 50 movecanvas                           % Move the child canvas across the
                                           % parent; damage occurs to the
                                           % parent.
```

The damage caused by moving BlackCanvas is illustrated in the following figure:

Figure 2-16 *Damage on unretained GrayCanvas after moving BlackCanvas*

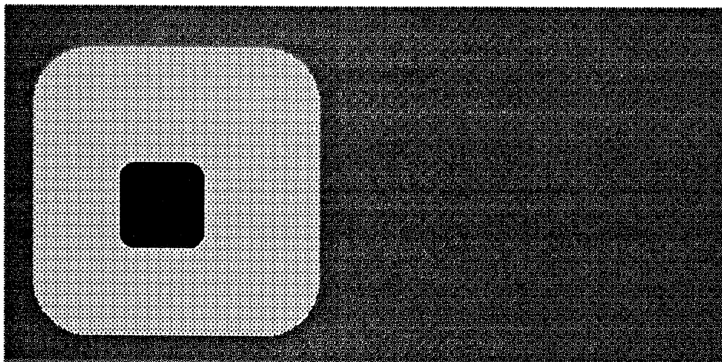


If the damaged parent canvas is repainted (to repair the damage) and then retained, the child canvas can be moved over its surface without damage occurring:

```
GrayCanvas setcanvas

GrayCanvas /Retained true put      % Retain GrayCanvas, which
.88 fillcanvas                     % causes damage where it is
                                   % obscured by BlackCanvas.
                                   % Repair damage by repainting.

BlackCanvas setcanvas              % Move the child across the retained
75 75 movecanvas                   % parent; no damage occurs.
```

The following figure illustrates BlackCanvas moved to its new position on undamaged GrayCanvas:

Figure 2-17 *No damage on retained GrayCanvas after moving BlackCanvas*



NOTE *When a canvas is made retained, the server sets to zero the value of the bits that represent the offscreen memory of the invisible portions of the canvas. The canvas then receives damage on its invisible areas and the client can repaint those areas. If the canvas is mapped to the screen after it is made retained and before the client repaints it, the previously invisible portions of the canvas will appear on the screen with whatever color is assigned to the pixel value of zero (usually*

white on a monochrome screen). Thus, you might see the image of a mapped retained canvas on the screen even before you draw in it.

Avoiding Canvas Damage with SaveBehind Canvases

The **SaveBehind** key of a canvas can be used to prevent damage from occurring to other canvases. When the key is set to **true**, the server saves the values of the pixels that the canvas obscures when it is mapped. Even if the pixels belong to unretained canvases, they can be restored directly to the screen when the **SaveBehind** canvas is unmapped. Note that **SaveBehind** does not prevent damage if the canvas is moved — only if it is unmapped.

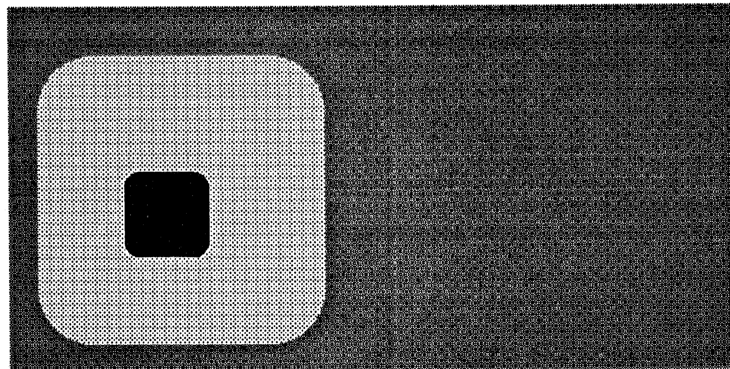
If drawing occurs on an area of a canvas while that area is obscured by a **SaveBehind** canvas, that area will receive damage; the server does not keep an updated record of the pixel values that the **SaveBehind** canvas obscures. Therefore, **SaveBehind** canvases are only useful if they are mapped for short periods of time. The **SaveBehind** key is useful for pop-up menus and other canvases that are small and are not required to be visible for long; when used with such canvases, the key can greatly enhance server performance.

The following example (a continuation of the example in the previous subsection) shows that damage occurs to unretained GrayCanvas when BlackCanvas is unmapped:

```
GrayCanvas /Retained false put        % Make the parent unretained.
BlackCanvas /Mapped false put         % Unmap the child.
```

The following figure illustrates that although BlackCanvas is unmapped, its image is still visible because GrayCanvas is damaged:

Figure 2-18 *Unretained GrayCanvas damaged by unmapping BlackCanvas*

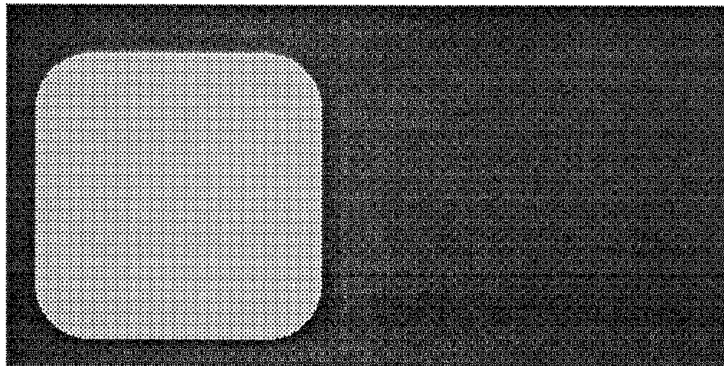


The following code repairs GrayCanvas and makes BlackCanvas a **SaveBehind** canvas. Then BlackCanvas is mapped and unmapped with no damage to GrayCanvas.

```
GrayCanvas setcanvas 0.88 fillcanvas    % Repair damage.
BlackCanvas /SaveBehind true put        % Specify use of SaveBehind.
BlackCanvas /Mapped true put            % Remap the child.
BlackCanvas setcanvas                   % Make the child current, paint
0 fillcanvas                            % it, and unmap it; no damage
BlackCanvas /Mapped false put           % occurs to the parent.

quit                                    % Quit psh.
```

The following figure illustrates that, this time, GrayCanvas is not damaged when BlackCanvas is unmapped:

Figure 2-19 *GrayCanvas not damaged by unmapping SaveBehind BlackCanvas*



2.5. Restricting the Drawing Area with the Canvas Clip

Each NeWS process has a *current clipping path* defined as part of its current graphics state. Drawing operations performed by a process are restricted to the area enclosed by the process' current clipping path.

Likewise, each NeWS canvas can have a *canvas clipping path* associated with it. Drawing operations performed on a canvas are restricted to the area given by the intersection of the canvas' clipping path, the process' current clipping path, and the canvas' shape. The canvas clipping path is typically used to limit the portion of a canvas that is painted during damage repair (by setting the canvas clipping path to be the path returned by **damagepath** before repairing the damage).

The following operators can be used to set and inspect a canvas' clipping path (for more detailed descriptions, see Chapter 10, "NeWS Operator Extensions"):

– **clipcanvas** –
This operator sets the clipping path of the current canvas to be the same as the current path; if the current path is empty, **clipcanvas** removes any existing clipping restriction of the current canvas.

– **clipcanvaspath** –
This operator sets the current path to be the same as the clipping path of the current canvas.

– eoclipcanvas –

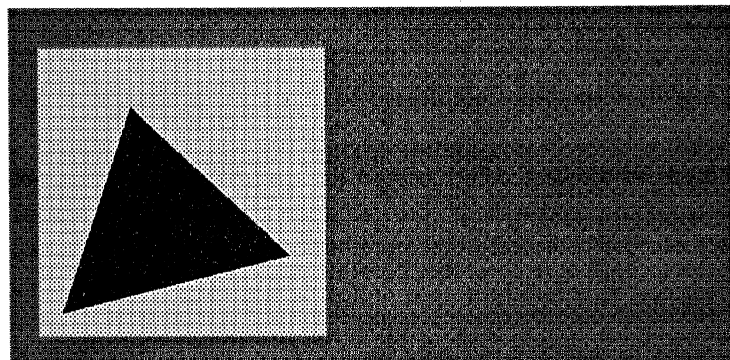
This operator is the same as **clipcanvas**, except that it uses the even-odd rule instead of the non-zero winding number rule to interpret the path.

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

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/FirstCanvas framebuffer            % Create FirstCanvas.
newcanvas def
25 25 translate
0 0 250 250 rectpath
FirstCanvas reshapecanvas
FirstCanvas /Retained false put
FirstCanvas /Mapped true put
FirstCanvas setcanvas 0.88 fillcanvas

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

clipcanvas                          % Set the canvas clipping path to the path.
0 fillcanvas                        % The fillcanvas operation is performed —
                                    % only the triangular path is filled.
```

When the above **fillcanvas** is executed, the area that is filled is the intersection of the current clipping path, the canvas shape, and the triangular canvas clipping path. In this case, that intersection is the entire interior of the triangular clipping path. The appearance of FirstCanvas is now as follows:

Figure 2-20 *Results of filling FirstCanvas after setting a canvas clipping path*

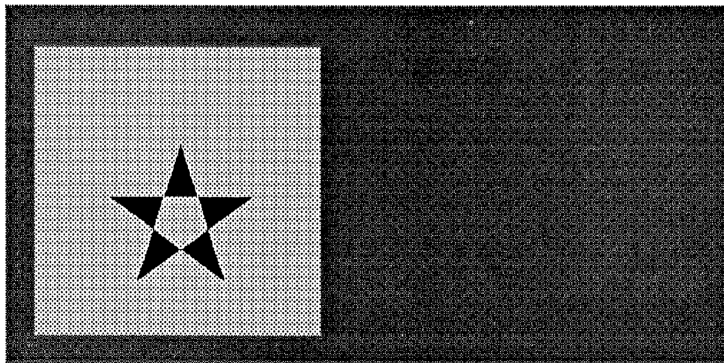


The next example (a continuation of the previous example) demonstrates the **eoclipcanvas** operator:

```
newpath clipcanvas                          % Remove the previous canvas
                                            % clipping path.
0.88 fillcanvas                             % Paint FirstCanvas gray.

/starpath {
    translate
    0 0 moveto
    4 {
        125 0 translate
        0 0 lineto
        -144 rotate
    } repeat
    closepath
} def

newpath 65 120 starpath eoclipcanvas        % Assign clipping path.
0 fillcanvas                                % Paint FirstCanvas black.

quit                                        % Quit psh.
```

The following figure illustrates the new appearance of FirstCanvas. Because this example uses **eoclipcanvas**, only the points of the star are filled.

Figure 2-21 *Results of eoclipcanvas*



2.6. Manipulating the Canvas Hierarchy

Each canvas has a key that points to its parent (**Parent**), a key that points to the sibling immediately above it (**CanvasAbove**), and a key that points to the sibling immediately below it (**CanvasBelow**). The **Parent** key establishes the parent/child relationships of the canvas hierarchy. The **CanvasAbove** and **CanvasBelow** keys effectively arrange a canvas' children into an ordered list from the bottom child to the top child. A canvas' position in the hierarchy can be manipulated by changing the value of any of these three keys. A canvas' children can also be rearranged in their sibling list with four NeWS operators.

This section discusses how to use these keys and operators to manipulate the canvas hierarchy.

Changing Sibling Relationships

When the **newcanvas** operator is executed, the newly created canvas becomes the top child of its parent. You can change a canvas' position in its sibling list by changing the value of its **CanvasAbove** or **CanvasBelow** key. The canvas dictionary keys that relate to sibling hierarchy are described below.

- **CanvasAbove** (read/write)

  The value of a canvas' **CanvasAbove** key is the canvas that is immediately above it in its sibling list; if no such canvas exists, the value of the key is **null**. This key can be set to any of the canvas' siblings. When the value is changed, the canvas is inserted in the list at a position directly below the specified sibling; the canvas does not change its (*x*, *y*) position on the screen, but the appearance of the canvas and its siblings changes to reflect their new overlapping relationships.

- **CanvasBelow** (read/write)

  The value of a canvas' **CanvasBelow** key is the canvas that is immediately below it in its sibling list; if no such canvas exists, the value of the key is **null**. This key can be set to any of the canvas' siblings. When the value is changed, the canvas is inserted in the list at a position directly above the specified sibling; the canvas does not change its (*x*, *y*) position on the screen, but the appearance of the canvas and its siblings changes to reflect their new overlapping relationships.

- **TopCanvas** (read-only)

  The value of a canvas' **TopCanvas** key is the canvas' top sibling. If the canvas has no siblings, the value is the canvas itself.

- **BottomCanvas** (read-only)

  The value of a canvas' **BottomCanvas** key is the canvas' bottom sibling. If the canvas has no siblings, the value is the canvas itself.

- **TopChild** (read-only)

  The value of a canvas' **TopChild** key is the canvas' top child or **null** if the canvas has no children.

Note that when you change the value of a canvas' **CanvasAbove** or **CanvasBelow** key, the server automatically changes the value of the other relevant canvas keys (**CanvasBelow** or **CanvasAbove**, **TopCanvas**, and **BottomCanvas**) to reflect the new sibling order. The server also makes any necessary adjustments to the keys of the other siblings in the list, and it updates the **TopChild** key of the parent if necessary.

In addition to changing the value of the **CanvasAbove** and **CanvasBelow** keys, you can also manipulate the sibling list with the following operators:

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

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

canvas x y **insertcanvasabove** –
Inserts the current canvas into the list at the position immediately above *canvas*. Also moves the current canvas to (*x*, *y*) relative to its parent's default coordinate system.

canvas x y **insertcanvasbelow** –
Inserts the current canvas into the list at the position immediately below *canvas*. Also moves the current canvas to (*x*, *y*) relative to its parent's default coordinate system.

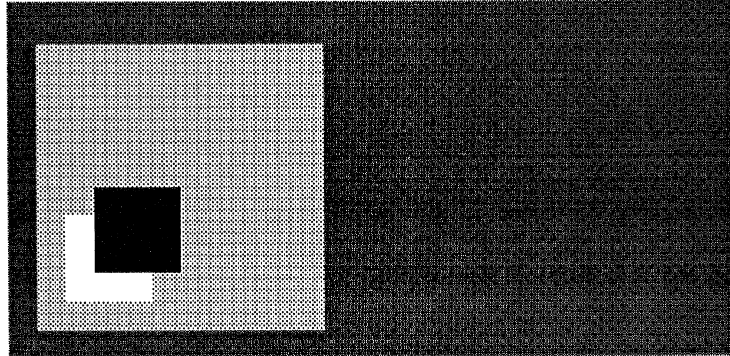
All the above operators cause adjustments to the keys of the affected siblings and parent. For the operators **insertcanvasabove** and **insertcanvasbelow**, the current canvas must be a sibling of the specified *canvas*.

The following example uses three canvases: FirstParent, WhiteCanvas, and BlackCanvas. WhiteCanvas and BlackCanvas are children of FirstParent. Because BlackCanvas is created after WhiteCanvas, it is placed at the top of the sibling list and thus obscures its sibling.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/MakeCanvas { % color dx dy x y w h parent => canvas
framebuffer setcanvas
newcanvas
5 1 roll newpath rectpath
dup reshapecanvas dup setcanvas
3 1 roll movecanvas
dup begin
    /Transparent false def
    /Retained true def
    /Mapped true def
end
exch fillcanvas
} def

/FirstParent                              % Create parent.
0.88 25 25 0 0 250 250 framebuffer MakeCanvas
def

/WhiteCanvas                              % Create one child.
1 25 25 0 0 75 75 FirstParent MakeCanvas
def

/BlackCanvas                              % Create another child.
0 50 50 0 0 75 75 FirstParent MakeCanvas
def                                       % BlackCanvas obscures
                                          % WhiteCanvas.
```

The newly created parent and children are illustrated in the following figure:

Figure 2-22 *BlackCanvas obscuring WhiteCanvas*

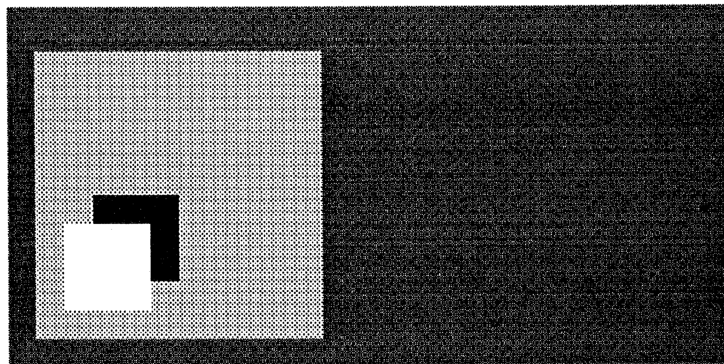


The following code uses **insertcanvasabove** to insert WhiteCanvas above BlackCanvas:

```
WhiteCanvas setcanvas                  % Insert WhiteCanvas above
BlackCanvas 25 25 insertcanvasabove    % BlackCanvas.
```

The appearance of the canvases is now as follows:

Figure 2-23 *WhiteCanvas made to obscure BlackCanvas*



The following example uses **canvastotop** to make BlackCanvas the top sibling again:

```
BlackCanvas canvastotop
```

The following code also makes BlackCanvas the top sibling, but it uses a canvas dictionary key instead of a canvas operator:

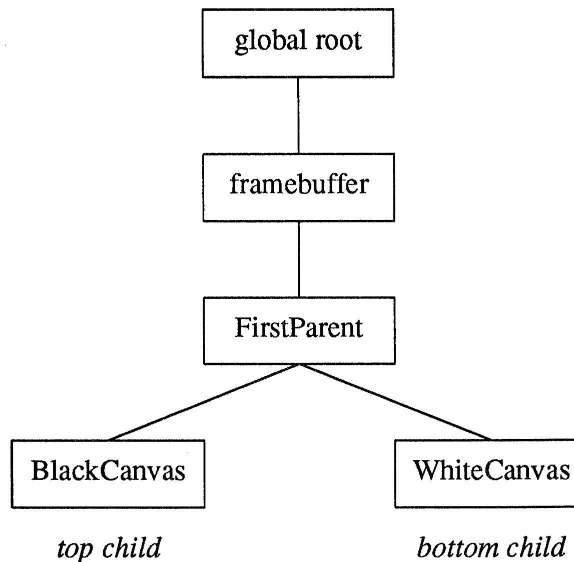
```
BlackCanvas /CanvasBelow WhiteCanvas put
```

Establishing a New Parent

You can specify a new parent for a canvas by setting the value of the **Parent** key in the **canvastype** dictionary. Note that you cannot make one of a canvas' descendants be its parent, nor can you reparent a canvas so that it moves from one framebuffer canvas' subhierarchy to another framebuffer canvas' subhierarchy.

The following example builds on the previous example to demonstrate changing a canvas' parent. At this point, the example's canvas hierarchy can be represented by the following tree:

Figure 2-24 *Canvas hierarchy*

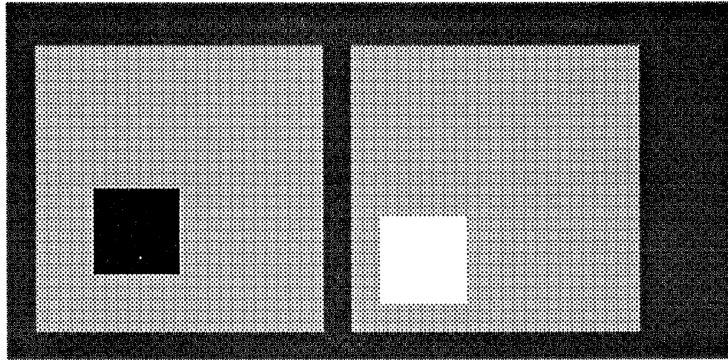


The following code creates a new canvas, SecondParent, that is a child of the framebuffer. The code then changes the parent of WhiteCanvas to be SecondParent:

```
/SecondParent                                % Create SecondParent.
0.88 300 25 0 0 250 250 framebuffer MakeCanvas
def

WhiteCanvas /Parent SecondParent put         % Make SecondParent the
                                             % parent of WhiteCanvas.

quit                                         % Quit psh.
```

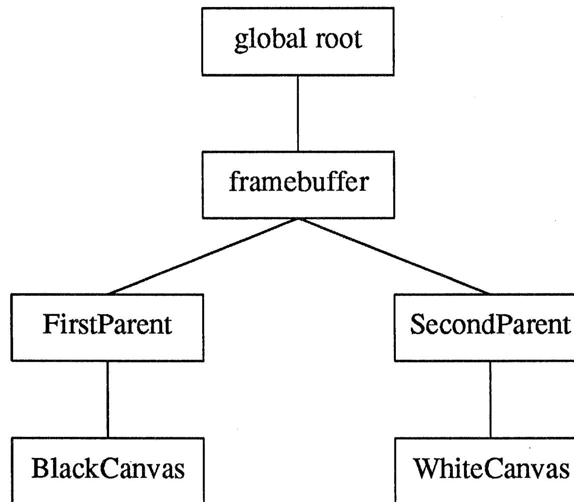
The canvases now appear as follows (note that WhiteCanvas is placed in the same position relative to SecondParent's upper-left corner as it had relative to FirstParent's upper-left corner):

Figure 2-25 *WhiteCanvas is now the child of SecondParent*



The new hierarchy is shown in the following figure:

Figure 2-26 *New canvas hierarchy*



2.7. Overlay Canvases

The server allows you to create *overlay canvases*. An overlay canvas, which can only be created over an existing non-overlay canvas, does not obscure the canvases beneath it. However, unlike transparent canvases, graphic objects drawn on an overlay appear on the overlay itself rather than on the canvas below. Thus, drawing in an overlay does not interfere with drawing in its associated canvas, and drawing in the canvas does not interfere with drawing in the overlay. An overlay is like a sheet of cellophane that floats over a canvas and all the canvas' children. The overlay is always the same size as the canvas that it overlays.

Overlays are intended for use in transient or animated drawing procedures. For example, they can be used to create "rubber-band" boxes, which expand or contract according to mouse movement when a user is resizing a window. In general, overlays are useful when you want to draw a temporary image over a canvas without having to repaint the canvas after you erase the temporary image.

You should not change the keys in an overlay's dictionary. For example, you should not attempt to map or unmap an overlay; an overlay assumes the mapped

state of its asociated non-overlay. To remove the images drawn in an overlay, you can use the **erasepage** operator. To destroy an overlay, you must remove all references to it, as you would for any other canvas.

Other features of overlays are as follows:

- Each non-overlay canvas (whether transparent or opaque) can possess one overlay canvas only. If a canvas possesses an overlay, any subsequent attempt to create an overlay of the canvas returns the existing overlay.
- An overlay canvas cannot receive any events. If you express interest on an overlay, the interest is placed on the pre-child interest list of the canvas over which the overlay was created. For a complete description of events and interests, see Chapter 4, "Events."
- An overlay never receives damage and, therefore, never requires repainting.
- An overlay cannot have a parent, nor can it have children.
- If an overlay's corresponding non-overlay canvas has children, these children may have their own overlays. A canvas' overlay appears above the overlays of the canvas' children.
- An overlay cannot be reshaped; attempting to reshape an overlay produces no result. An overlay always has the shape of its associated non-overlay canvas.
- An overlay cannot be possessed by more than one non-overlay, nor can it change owners.

Creating and Using Overlays

The following operator creates an overlay canvas:

canvas **createoverlay** ocanvas

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

The **createoverlay** operator is demonstrated in the following example. This code creates a canvas and an associated overlay. A grid is drawn on the overlay, and then a simple picture of a house is drawn on the canvas beneath the overlay, using the grid as a guide.

You may notice that the overlay flashes on the screen when you move the mouse or type; this flashing is an artifact of the way overlays are implemented on some machines. You may also notice some decrease in performance when typing this example. The flashing and performance problems are discussed in the following subsection, "Restrictions for Drawing on Overlays."

Because of the flashing problems, you might prefer to type the code shown in the next box into a file, enter an executive `psh` session, and then load your file into the `psh` session with the following command:

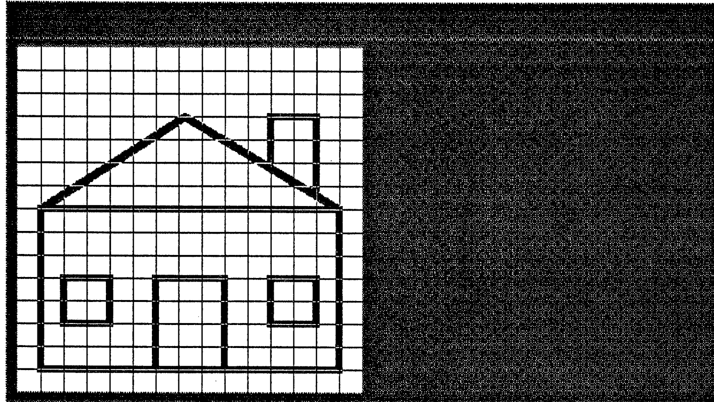
```
(filename) LoadFile
```

Loadfile is a utility provided by the POSTSCRIPT language extensibility files.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/MyCanvas framebuffer                       % Create MyCanvas.
newcanvas def
0 0 300 300 rectpath MyCanvas reshapecanvas
MyCanvas /Mapped true put
MyCanvas setcanvas 1 fillcanvas
10 10 movecanvas

/OverCanvas MyCanvas createoverlay def      % Create an overlay.
OverCanvas setcanvas
0 setgray
20 20 280 {                                 % Draw a grid on overlay.
    dup 0 moveto dup 300 lineto stroke
    dup 0 exch moveto 300 exch lineto stroke
} for

MyCanvas setcanvas
7 setlinewidth

20 20 260 140 rectpath stroke               % House.

20 160 moveto 145 240 lineto                % Roof.
280 160 lineto stroke

120 20 moveto 120 100 lineto                % Door.
180 100 lineto 180 20 lineto stroke

40 60 40 40 rectpath stroke                 % Left window.

220 60 40 40 rectpath stroke                % Right window.

220 200 moveto 220 240 lineto               % Chimney.
260 240 lineto 260 175 lineto stroke
```

The following figure illustrates the grid on the overlay and the house image on the canvas beneath the overlay:

Figure 2-27 *A canvas and its overlay*



An overlay has the same mapped status as the canvas it overlays. The following example unmaps and maps MyCanvas, demonstrating that the overlay is also unmapped and mapped.

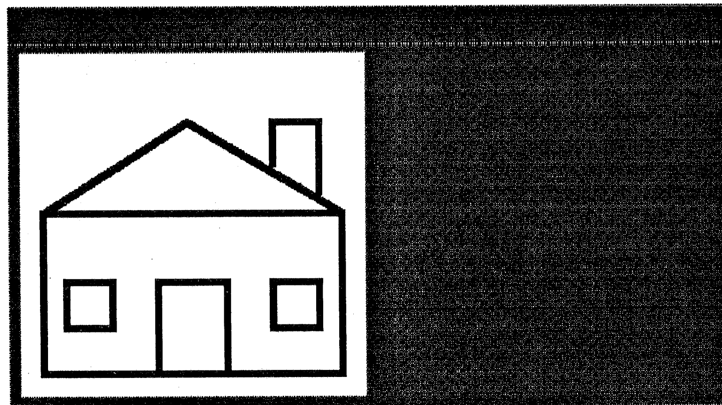
```
MyCanvas /Mapped false put          % Both MyCanvas and
                                    % overlay disappear.
MyCanvas /Mapped true put           % Both MyCanvas and
                                    % overlay appear again.
```

When you no longer want the overlay to be visible, you simply erase the drawing that it contains. The following example erases the grid from OverCanvas:

```
OverCanvas setcanvas
erasepage                           % Erase grid from OverCanvas.

quit                                % Quit psh.
```

The following figure illustrates the erased overlay and the canvas underneath:

Figure 2-28 *A canvas and its erased overlay*



Restrictions for Drawing on Overlays

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

The current color is usually ignored when drawing operations are performed on overlays. This behavior is deliberate; it allows the implementation of overlays to vary on different kinds of hardware.

On the machine that was used to generate the house example above, the house canvas was XOR'd to produce the colors used for each bit of its overlay. On a monochrome screen, this XOR procedure results in an overlay that always uses the opposite color of the image underneath. Where a grid line lay over the white background, it was painted black; where a grid line lay over a black line of the house, it was painted white (see Figure 2-27 in the previous subsection). Overlays may be implemented differently on other machines.

After an image is drawn in an overlay, the overlay's image may flash when any portion of the screen is repainted. For example, flashing may occur when the mouse cursor moves across the screen or when input is typed into a window. Flashing may also occur when a canvas that owns an overlay is damaged and repainted. To avoid problems with flashing, images drawn in an overlay should not be maintained for too long. This restriction limits the use of overlays to special situations such as implementing rubber-band boxes.

The Framebuffer Overlay

When the server is initialized, the framebuffer canvas and an associated overlay are created. The framebuffer's overlay is named **fboverlay**. You can use **fboverlay** in the same way as any other overlay canvas. NeWS applications commonly use **fboverlay** to implement their rubber-band boxes.

2.8. Canvases, Files, and Imaging Procedures

You can save in a file the image drawn on a canvas and read it back into a canvas object. You can also image canvas objects onto the current canvas. This section describes the operators that you can use to accomplish these tasks.

Writing Canvases to Files

The following operator writes a canvas image to a file:

file *or* string **writecanvas** –

This operator writes the current canvas image to a raster file. The raster file can be specified either as a file or as a string that is the name of a file in the server's file name space. The operator creates a raster file that contains an image of the region outlined by the current path in the current canvas. If the current path is empty, the whole canvas is used.

The **writecanvas** operator uses the non-zero winding number rule. To write a canvas image to a file using the even-odd rule, use **eowritecanvas**. See the *POSTSCRIPT Language Reference Manual* for information about these rules. Note that an unretained rooted canvas should be mapped before using **writecanvas** or **eowritecanvas**.

The following operator writes a region of the screen, outlined by the current path in the current canvas, to a file:

file *or* string **writescreen** –

This operator is the same as **writecanvas** except that if the current canvas is partially obscured by one or more canvases that lie on top of it, **writescreen** includes the overlapping canvases in the image. Thus, a screendump can be performed by setting the current canvas to be the framebuffer canvas and then executing **writescreen**.

The **writescreen** operator uses the non-zero winding number rule. To perform the same operation using the even-odd rule, use **eowritescreen** instead.

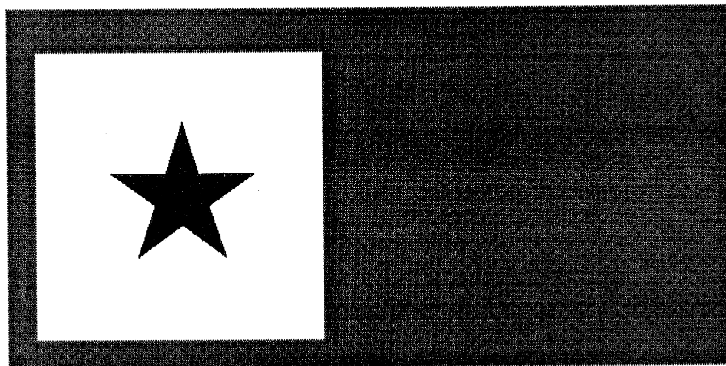
The **writecanvas** operator is demonstrated by the following example. This example creates a canvas named StarCanvas and paints a star on it. The canvas image is then written to a file named starfile.

NOTE **writecanvas** *and* **writescreen** *store images in a fixed-resolution raster image format, not as s executable files.*

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/StarCanvas framebuffer                    % Create a canvas.
newcanvas def
25 25 translate
0 0 250 250 rectpath
StarCanvas reshapecanvas
StarCanvas /Retained true put
StarCanvas setcanvas
1 fillcanvas                               % Paint the canvas.
StarCanvas /Mapped true put                % Map the canvas.

/starpath {
    translate
    0 0 moveto
    4 {
        125 0 translate
        0 0 lineto
        -144 rotate
    } repeat
    closepath
} def

65 145 starpath clipcanvas
0 fillcanvas                               % Paint a star on the canvas.
(starfile) writecanvas                     % Write the canvas to a file.
```

The canvas created in the above code is illustrated in the following figure:

Figure 2-29 *StarCanvas*



In the next two subsections, starfile is read back into a canvas and then imaged to the screen.

NOTE *The raster files created by* **writecanvas, eowritecanvas, writescreen,** *and* **eowritescreen** *are rectangular. If the canvas that is written to the file is not rectangular, the bits between the canvas' bounding box and the canvas' shape are given 0 values.*

Reading Canvases from Files

A file created with **writecanvas, writescreen, eowritecanvas,** or **eowritescreen** can be read back into a NeWS canvas object. The following operator creates a canvas from such a file:

string *or* file **readcanvas** canvas

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

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

The **readcanvas** operator is demonstrated by the following example, which reads the canvas saved in the previous example into a new canvas named FileCanvas:

```
/FileCanvas (starfile) readcanvas def
```

A canvas read with **readcanvas** cannot be mapped to the display. To image the canvas to the screen, you must use the **imagecanvas** operator, described in the next section. If you change the value of an irrelevant key of an unrooted canvas (such as the **Mapped** key), the change has no effect; if you attempt to execute an irrelevant operator (such as **movecanvas** or **reshapecanvas**) you will receive an error message.

Imaging a Canvas to the Screen

The following operator paints a canvas' image onto the current canvas:

canvas **imagecanvas** –

This operator paints *canvas* onto the current canvas. The entire source canvas is imaged onto the current canvas in such a way that the unit square of the source canvas is mapped onto the unit square of the current canvas. This operator is similar to the **image** operator provided by the POSTSCRIPT language except that the image comes from a canvas instead of a POSTSCRIPT language procedure.

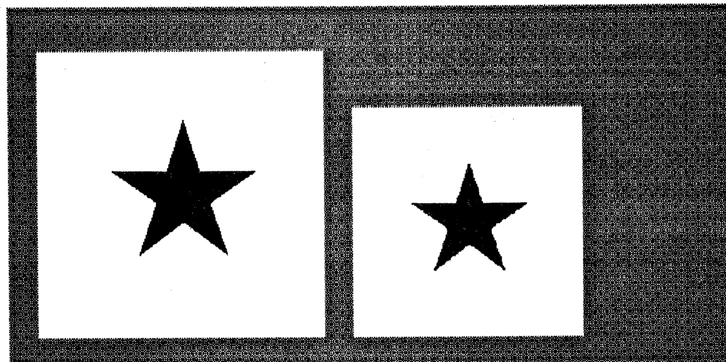
Any type of canvas can be used as a source for the **imagecanvas** operator: a rooted canvas, an unrooted canvas created with **readcanvas,** or an unrooted canvas created with **buildimage.** When you use **imagecanvas,** you must consider the default coordinate systems of both the source canvas and the destination canvas to ensure that the resulting image has the desired scale and position. If the source canvas is rooted, its default transformation matrix was assigned when it was last reshaped. If the source canvas is an unrooted canvas created with **readcanvas,** its default transformation matrix was assigned by **readcanvas** to map the entire canvas to the unit square. If the source canvas is an unrooted canvas created with **buildimage,** its default transformation matrix was assigned as the matrix argument to **buildimage.**

The following example creates a new canvas named SecondCanvas and sets it to be the current canvas. It then scales up the CTM by a factor of 200 so that the subsequent **imagecanvas** fills all of SecondCanvas. The canvas that is imaged is FileCanvas, which is an unrooted canvas that was created with **readcanvas** in the previous example.

```
framebuffer setcanvas
/SecondCanvas framebuffer                    % Create SecondCanvas.
newcanvas def
300 25 translate
newpath 0 0 200 200 rectpath
SecondCanvas reshapecanvas
SecondCanvas /Mapped true put
SecondCanvas setcanvas 0.88 fillcanvas       % Paint SecondCanvas gray.

200 200 scale
FileCanvas imagecanvas                       % Image FileCanvas onto
                                             % SecondCanvas.
```

The following figure illustrates FileCanvas imaged onto SecondCanvas:

Figure 2-30 *FileCanvas imaged onto SecondCanvas*

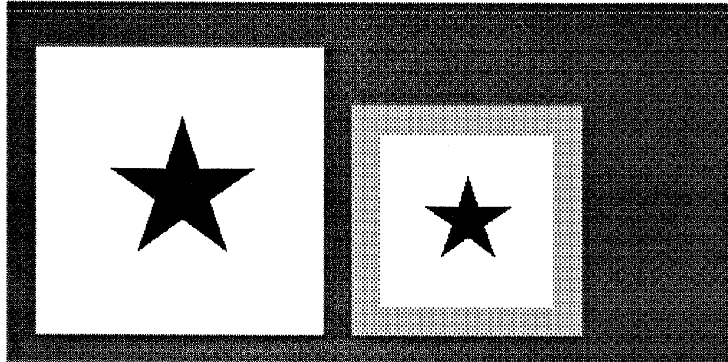


The next example demonstrates that a rooted canvas can be imaged onto the current canvas. In this case, StarCanvas is imaged directly onto SecondCanvas. The CTM is translated and scaled to map the image onto the center of SecondCanvas, leaving a border around the image.

```
SecondCanvas setcanvas
0.88 fillcanvas                              % Repaint SecondCanvas gray.
25 25 translate
0.6 0.6 scale
StarCanvas imagecanvas                       % Image StarCanvas directly
                                             % onto SecondCanvas.

quit                                         % Quit psh.
```

The new appearance of SecondCanvas is shown in the following figure:

Figure 2-31 *StarCanvas imaged onto SecondCanvas*



The following operator can be used to paint a 1-bit deep canvas onto the current canvas, using a boolean argument and the current color to specify how the canvas should be painted:

boolean canvas **imagemaskcanvas** –

This operator paints *canvas* onto the current canvas, just as **imagecanvas** does. If *boolean* is **true, imagemaskcanvas** paints the 1 bits with the current color; if *boolean* is **false, imagemaskcanvas** paints the 0 bits with the current color. Thus, the operator essentially defines a mask through which color is painted. The operator is only valid for 1-bit deep source canvases.

For an example of **imagemaskcanvas,** see the example of **buildimage** given in the following section.

NOTE *In the current implementation, the* **imagecanvas** *and* **imagemaskcanvas** *operators paint the region within the source canvas' bounding box, rather than painting just the canvas' interior. This difference becomes apparent if you image a non-rectangular canvas.*

If you image an unretained canvas that is non-rectangular, the bits outside the canvas' shape but inside the canvas' bounding box are imaged with whatever color they have on the screen. If you image a retained canvas that is not rectangular (either rooted or unrooted), the bits outside the canvas' shape but inside the canvas' bounding box are imaged with whatever color is assigned to 0 (usually white on monochrome screens); these bits were assigned a 0 value when the canvas was made retained.

If you want to omit the area between the canvas' shape and its bounding box, simply clip to the canvas' shape when you image onto the current canvas.

Building a Canvas Image

The following operator builds an image that is stored in a canvas object:

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

The NeWS operator **buildimage** provides functionality similar to that of the **image** operator provided by the POSTSCRIPT language. The **buildimage** operator uses the binary representation of a specified string to create a sampled image as a

canvas object. The canvas object is retained, has no parent, and is not a part of the canvas hierarchy (it is an unrooted canvas). The canvas cannot be mapped; it can be imaged to the screen with the **imagecanvas** operator, or it can be written to a file with the **writecanvas** operator.

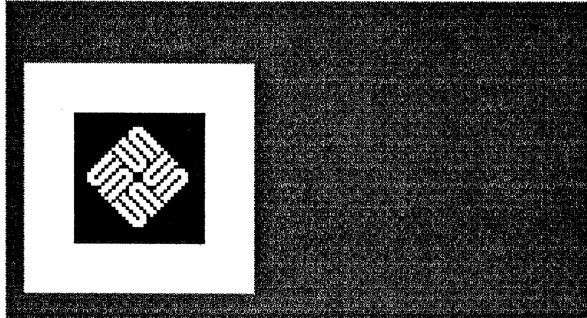
The *width*, *height*, *bits/sample*, and *proc* arguments are the same as for the POSTSCRIPT language **image** operator. The *matrix* argument defines the default coordinate system of the canvas. The arguments to the **buildimage** operator are described fully in Chapter 10, "NeWS Operator Extensions."

To create an empty unrooted canvas, you can give a null procedure to the **buildimage** operator. You can draw or image to the resulting canvas as with any other offscreen canvas. The following example uses **buildimage** to construct an image. The image is stored in a canvas object named source and is then painted onto the current canvas, dest, with the **imagemaskcanvas** operator.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/dest framebuffer newcanvas def
0 0 200 200 rectpath dest reshapecanvas
25 25 dest movecanvas
dest /Mapped true put
/source 38 38 1 [38 0 0 -38 0 38] {
< FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
  FFFF8FFFFF FFFF07FFFF FFFF23FFFF FFFD11FFFF
  FFF888FFFF FFF4447FFF FFE2223FFF FFD1115FFF
  FF88888FFF FF144517FF FE222623FF FC470447FF
  F8888888FF F11071107F F22272227F F04470447F
  F8888888FF FF110711FF FE232223FF FF451147FF
  FF88888FFF FFD4445FFF FFE2223FFF FFF1117FFF
  FFF888FFFF FFFC45FFFF FFFE27FFFF FFFF07FFFF
  FFFF8FFFFF FFFFFFFFFF FFFFFFFFFF FFFFFFFFFF
  FFFFFFFFFF FFFFFFFFFF >
} buildimage def

dest setcanvas
1 fillcanvas
0 setgray
/size 38 3 mul def
200 size sub 2 div dup translate          % Center the image.
size dup scale
true source imagemaskcanvas
```

The appearance of dest is now as follows:

Figure 2-32 *Image built with* **buildimage** *and imaged with* **imagemaskcanvas**

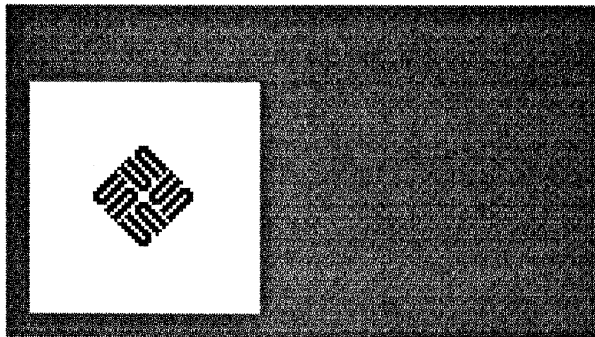


The following example shows how the image changes if the 0 bits, instead of the 1 bits, are painted black:

```
1 fillcanvas
0 setgray
false source imagemaskcanvas

quit                                    % Quit psh.
```

The following figure illustrates the new image:

Figure 2-33 *Image with 0 bits painted black*



2.9. Cursors

The **canvastype** dictionary contains a **Cursor** key, which specifies the cursor object that is used whenever the mouse is positioned over the canvas. When a canvas is created with **newcanvas**, its **Cursor** value is initially **null**; unless the canvas' **Cursor** key is given some non-null value, its parent's cursor is displayed whenever the mouse is over the canvas.

Cursor Objects

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

their characters.

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

A cursor object is implemented as a dictionary with the following six keys:

- **CursorChar** and **MaskChar** (read-only)

  The integer that corresponds to the character used for the cursor image and mask image, respectively.

- **CursorFont** and **MaskFont** (read-only)

  The font used for the cursor image and mask image, respectively.

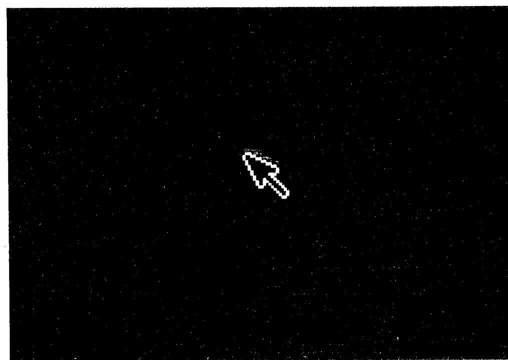
- **CursorColor** and **MaskColor** (read/write)

  The color used to paint the cursor image and mask image, respectively.

Standard Cursors

The server provides a set of standard cursor and mask characters in a special font named **cursorfont**. For example, the default cursor for the framebuffer canvas is an arrow that points up and to the left. The following figure illustrates the framebuffer's default cursor:

Figure 2-34 *The framebuffer's default cursor*



The framebuffer's default cursor is created with a cursor character named **basic** and a mask character named **basic_m**. These character names are keys in a dictionary named **cursordict**; the value of each key in **cursordict** is the integer that corresponds to that character in **cursorfont**. (The names in **cursordict** are easier to remember than the integers associated with the characters in **cursorfont**; also, the integers are subject to change but the names are not.) For example, the integer associated with **basic** is 0. Therefore, the default value of the framebuffer's **CursorChar** key is 0.

The **cursordict** dictionary is located in the file `$OPENWINHOME/etc/NeWS/cursor.ps`. The `cursor.ps` file is loaded when the server is initialized. See Chapter 9, "NeWS Type Extensions," for more information about the standard cursors provided by **cursorfont**.

Changing a Canvas' Cursor

To change a canvas' cursor, you must create the new cursor object and then assign that cursor to the canvas' **Cursor** key. You can create a new cursor object with any of the characters in **cursorfont** or with characters that you create yourself. The **newcursor** operator creates a new cursor object from existing cursor and mask characters, as described below:

cchar mchar font **newcursor** cursor
cchar mchar cfont mfont **newcursor** cursor

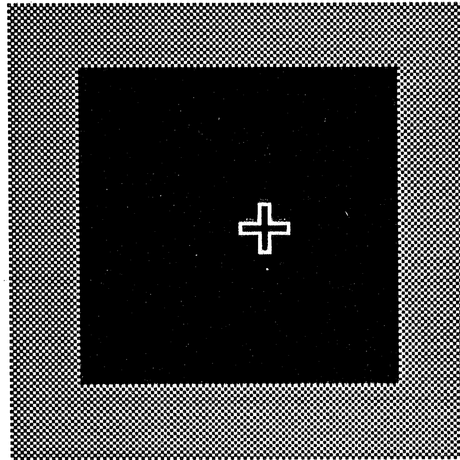
The **newcursor** operator creates an object of type **cursor**. Two syntactic forms can be used. With the first form, a cursor is constructed using the cursor character *cchar* and the mask character *mchar*; both are selected from *font*. With the second form, a cursor is constructed using *cchar* from the font *cfont* and *mchar* from the font *mfont*. In both forms, the new cursor is initialized with a **CursorColor** value of black and a **MaskColor** value of white.

The following example creates a canvas, MyCanvas, that is a child of the framebuffer canvas. The default cursor of MyCanvas is the same as the default cursor of the framebuffer canvas. The example then creates a new cursor and sets it to be MyCanvas' cursor.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

/MyCanvas framebuffer newcanvas def
0 0 100 100 rectpath MyCanvas reshapecanvas
MyCanvas /Mapped true put
MyCanvas setcanvas 0 fillcanvas          % Paint MyCanvas black.

25 25 movecanvas

% Move the cursor over MyCanvas.
% See the default cursor inherited
% from the framebuffer.

cursordict begin
MyCanvas /Cursor                         % Assign a new cursor.
xhair xhair_m cursorfont newcursor put
end

% Move cursor over MyCanvas.
% See the new cursor.
```

MyCanvas and its new crosshair cursor are illustrated in the following figure:

Figure 2-35 *MyCanvas with a crosshair cursor*



Changing a Cursor's Color

You can change a cursor's color by changing the value of its **CursorColor** and **MaskColor** keys, which default to black and white respectively. The cursor colors are not guaranteed to be displayed as requested, but the mask and cursor images will always be given contrasting colors.

The following example changes the value of **CursorColor** and **MaskColor** for MyCanvas' crosshair cursor; the example reverses the cursor colors so that the cursor image is white and the mask image is black.

```
% Move the cursor over MyCanvas.
% By default, the mask image
% is white and the cursor
% image is black.

1 fillcanvas                          % Paint MyCanvas white.

MyCanvas /Cursor get                  % Get MyCanvas' cursor.
begin
/CursorColor 1 1 1 rgbcolor def       % Change the colors.
/MaskColor 0 0 0 rgbcolor def
end

% Move cursor over MyCanvas.
% The cursor colors have reversed.

quit                                  % Quit psh.
```

2.10. Using Multiple Screens

This section describes the characteristics of the global root canvas and discusses some aspects of programming with multiple screens. To learn how to install multiple screens to run with your server, see the *X11/NeWS Server Guide*. If you are running the server with only one display screen, you do not need the information in this section.

The Global Root Canvas

The global root canvas, which is the root of the server's canvas hierarchy, is created when the server is initialized. The global root canvas is a transparent, mapped, unretained, very large canvas with **null** as its parent. The coordinate system of the global root canvas has its origin in the center of the canvas, the positive *y* axis extending vertically upward, the positive *x* axis extending horizontally to the right, and units of pixels in both coordinate directions. The global root canvas has dimensions of 32767 by 32767 pixels. (In the future, the server may change from units of pixels to units of 1/72 of an inch, consistent with the standard POSTSCRIPT language.)

Creating the Framebuffer Canvases

Each display screen has an associated framebuffer canvas that is a child of the global root canvas. When the server is initialized, the **createdevice** operator is called to create a canvas that covers the entire background of the initial display screen. The initial display screen is given by the value of the `FRAMEBUFFER` environment variable, which defaults to `/dev/fb`. You can create background canvases for additional screens with the **createdevicecanvas** utility:

string **createdevicecanvas** canvas *or* boolean

This utility creates and initializes additional framebuffer canvases. The *string* argument, which is system dependent, indicates the framebuffer canvas that is to be initialized (for example, `/dev/cgtwo0`). If **createdevicecanvas** fails to create the framebuffer canvas, it returns **false**. If it succeeds, it returns the framebuffer canvas. This utility should only be used during system initialization (for example, from a `.startup.ps` file).

The placement of your framebuffer canvases on the global root canvas should closely parallel the placement of your display screens on your desktop; the relative layout of the framebuffer canvases on the global root canvas determines how the mouse will track from one screen to another. You can use **movecanvas** or **reshapecanvas** on your framebuffer canvases (in fact, you can use any canvas operator on a framebuffer canvas). For more information about positioning your framebuffer canvases on the global root canvas, see the *X11/NeWS Server Guide*.

Allowed Operations for the Global Root Canvas

You can use the following keys of the global root canvas:

EventsConsumed	
Interests	
Parent	(getting)
TopChild	(getting and setting)
TopCanvas	(getting)
BottomCanvas	(getting)
CanvasAbove	(getting)
CanvasBelow	(getting)

You can use the following operators on the global root canvas:

setcanvas
currentcanvas
clippath
clipcanvaspath
reshapecanvas

Benign Operations for the Global Root Canvas

The following operations have no visible effect when used on the global root canvas, but they will not generate any errors:

Any graphics operation
Any damage recovery operation
Any operation that does not return a value, but draws to the screen

Disallowed Operations for the Global Root Canvas

You will recieve on `invalidaccess` error if you try to use any of the following keys of the global root canvas:

Parent	(setting)
TopCanvas	(setting)
BottomCanvas	(setting)
CanvasAbove	(setting)
CanvasBelow	(setting)
Mapped	
Transparent	
Retained	
SaveBehind	
Color	

You will receive an `invalidaccess` error if you try to perform any of the following operations on the global root canvas:

createoverlay
readcanvas
writecanvas
writescreen
eowritecanvas
eowritescreen
newcanvas
canvastotop
canvastobottom
insertcanvasabove
insertcanvasbelow
Any operation that returns a value and is not listed as allowed or benign

3

Processes

The X11/NeWS server maintains a set of simultaneously executing *lightweight processes*. A lightweight process is not a UNIX process; it is a process that resides in the server's address space and is scheduled to be run by the server. Lightweight processes are also known as *NeWS processes*.

Each NeWS process is an individual thread of control with its own graphics context, graphics state context, dictionary stack, execution stack, and operand stack. Each process can perform operations on the display and can receive events from the system (such as keyboard and mouse events) or from another NeWS process. Many processes can be created with relatively little overhead.

When the server first starts to run, it creates a single process that executes the startup file. At this time, code may be downloaded into the server and many more NeWS processes may start. Some NeWS processes communicate with client processes. Each connection to the server obtains its own NeWS process.

A new process is created with the **fork** operator; the newly created process is a *child* of the process that executed **fork**. A child process inherits its parent's dictionary stack, operand stack, and graphics state. Although a child process starts out with the same name space as its parent, each process can control the extent to which its name space is shared by pushing and popping dictionaries to and from its stack. When a child process is created, it is put in the same *process group* as its parent. Once created, the child process is not dependent on its parent for any resources. A process can be moved to its own, new process group if desired, and if the parent process dies for some reason, the child continues.

The server currently uses a non-preemptive round-robin scheduling policy. Processes must block periodically to allow other processes to run; if a process runs for more than 15 seconds without pausing, the server suspends the process and allows the next process to run. The server may adopt a preemptive scheduling policy in the future.

A NeWS process can kill its child processes, or it can wait for them to die and obtain a return value from them. A process can temporarily suspend itself or another process. The process that is currently running is known as the *current process*.

Monitor objects are provided for situations that require synchronization. For example, monitors should be used when writing replies from the server to the client.

This chapter describes NeWS processes and process operations.

The processtype Extension

Each NeWS process is an object of type **processtype**, which is a NeWS extension to the POSTSCRIPT language. Each **processtype** object can be accessed as a POSTSCRIPT language dictionary. A process dictionary includes keys that describe the following properties (the keys are listed in parentheses):

- The process stacks (**DictionaryStack, Execee, ExecutionStack, OperandStack, SendContexts, SendStack**)
- The process name (**ProcessName**)
- The process execution state (**State**)
- Standard files associated with the process (**Stdout, Stderr**)
- The process scheduler priority (**Priority**)
- The interest list of the process (**Interests**)
- Error information (**$error, errordict, ErrorCode, ErrorDetailLevel**)
- The process **bind** mode (**BindOverride**)
- The process packed array mode (**PackedArrays**)

Many of the process keys are described in this chapter. All of the NeWS types and their associated dictionaries are described in Chapter 9, "NeWS Type Extensions."

Process Operators

The NeWS language includes operator extensions to be used on processes. The process operators provide the following functionality (the operator names are listed in parentheses):

- Creating a new process or processgroup (**fork, newprocessgroup**)
- Controlling process execution (**breakpoint, continueprocess, pause, suspendprocess, waitprocess**)
- Destroying processes and processgroups (**killprocess, killprocessgroup**)
- Returning an array of processes or process groups (**getprocesses, getprocessgroup**)
- Returning the current process (**currentprocess**)
- Creating and using monitor objects (**createmonitor, monitor, monitorlocked**)
- Producing the process **$error** dictionary (**defaulterroraction**)
- Clearing the process **send** stack (**clearsendcontexts**)

Most of the process operators are described and demonstrated in this chapter. For a description of all NeWS operators, see Chapter 10, "NeWS Operator Extensions."

3.1. Basic Process Operations

This section describes basic process operations such as creating, pausing, stopping, restarting, and destroying processes. The examples use the psh command to establish a connection with the server. Some of the examples are interactive sessions with psh; other examples specify a code file as an argument to the psh command. Interactive sessions with psh or with the operating system shell are shown in gray boxes. Code examples that are meant to be typed into files are shown in plain boxes.

Some of the interactive examples are continuations of previous examples; you can tell that an example is a continuation if it does not start a new psh session at the top of the example code. You must type all the code sequentially from the start to the quit of each interactive psh session. For a complete description of the psh facility, see the psh manual page in the *X11/NeWS Server Guide.*

Establishing a Client Connection Process

When the server first starts, it creates a process that executes the initialization files. All other NeWS processes are descendants of this first process. One of the processes that the server forks is a listener process that listens for NeWS client connection requests. When a client connection is established, the listener process forks a process to serve that client, and it also creates a connection file associated with the newly forked process; the client's connection process executes the NeWS code that the client sends to the connection file. All processes forked by the client's code are children of its connection process. Thus, NeWS processes are arranged in a hierarchy. The main significance of the process hierarchy is that each child process inherits its parent's environment.

A C client program can establish a connection to the server with the ps_open_PostScript function (see Chapter 6, "C Client Interface"). If you want to execute pure NeWS code, you can use the psh command to establish a connection to the server.

When you execute the psh command, the server's NeWS listener process forks a lightweight process for the psh connection. If you give a filename as an argument to the psh command, the lightweight process executes the code that is contained in the file. If you enter an interactive psh session, the lightweight process executes code that you type or code that you load from a file with the **LoadFile** procedure (see the description of **LoadFile** in Chapter 11, "Extensibility through NeWS Procedure Files").

The following example establishes an interactive session with the server, creating a lightweight process associated with the psh connection:

```
% psh
executive
Welcome to X11/NeWS Version 2.1
```

The lightweight process associated with a psh session exits when the **quit** operator is executed or when the process encounters an EOF. When the lightweight process exits, the reference count on the associated connection file usually drops to zero, causing the connection file to be closed and destroyed. (When the total number of references to an object is zero, the server destroys the object and

reclaims its memory; see Chapter 8, "Memory Management," for details.) When the connection file is closed, the psh program exits, returning you to the operating system prompt.

Returning the Current Process

The following operator returns the current process:

– **currentprocess** process

The **currentprocess** operator places the current process object onto the operand stack.

The following example returns the current process, which is the process created previously with the psh command.

```
currentprocess ==
process (0x25c088, 'quakes NeWS client', runnable, '==')
```

Notice that four items are printed: a unique process identification number, the process name, the process state, and the object that is on top of the process execution stack. The name of a client connection process defaults to the name of the host. In this case, the process name defaults to `quakes NeWS client` because the host's name is `quakes`. The process state can be one of eight values, as described in the subsection "Examining the Process Execution State." This process has a **runnable** state, which indicates that the process is currently running or is scheduled to be run.

Examining the Process Stacks

Each NeWS process has an operand stack, an execution stack, a dictionary stack, and a graphics state stack. These four process stacks are described in detail in the *PostScript Language Reference Manual.* This section demonstrates how to access these stacks in a NeWS process.

Operand Stack

The following example puts some objects on the operand stack of the psh process created previously, and then it prints the contents of the operand stack with the POSTSCRIPT language **pstack** operator:

```
109 (mystring) /myname
pstack
109 (mystring) /myname
```

The following example also prints the contents of the operand stack, but this example uses the process **OperandStack** key instead of the **pstack** operator:

```
currentprocess /OperandStack get ==
[ 109 (mystring) /myname ]
```

Execution Stack

The value of a process' **ExecutionStack** key is the entire execution stack of the process. The value of a process' **Execee** key is the top item on the process' execution stack. These read-only keys can be useful when debugging.

Dictionary Stack

You can view the dictionary stack of a process by printing the contents of its **DictionaryStack** key. If a dictionary in the stack is very large, the == operator only prints part of its contents. For example, the == operator only prints some of the many entries in **systemdict**. (If you need to see all the entries in a large dictionary, you can write a procedure that uses the POSTSCRIPT language **forall** operator to print all the entries.)

The following example defines a procedure in the process' **userdict**; the procedure, named average, computes the average of two numbers. The example uses the procedure to compute the average of 4 and 2. Then the example prints the contents of the dictionary stack using the **DictionaryStack** key.

```
/average { % num num => –
   add 2 div =
} def
4 2 average
3.0

currentprocess /DictionaryStack get ==
[dict[....]
dict[ /OriginatingHost: (quakes)
/average: {'add' 2 'div' '='}]]
```

The first dictionary is the **systemdict**, and the second dictionary is the process' **userdict**. You can see that the previously defined average procedure is defined in the **userdict**. For brevity, the entries in **systemdict** are not shown in the box above but, when you try the example, you will see that some of the **systemdict** entries are printed. You will also notice that the **userdict** contains a key named **OriginatingHost** that contains the host's name; this key is automatically provided for a client connection process.

Graphics State Stack

No NeWS operator or process key returns the entire graphics state or graphics state stack. Instead, individual components of the graphics state can be set and inspected with POSTSCRIPT language operators and NeWS operator extensions. A list of the NeWS graphic state operators and their syntax is provided in Appendix A, "NeWS Operators." (The graphics state operators are listed in the miscellaneous category.)

The graphics state of the current process can be saved with the POSTSCRIPT language **gsave** operator; **gsave** places the current graphics state on the process' graphics state stack. The saved graphics state can be restored at a later time by executing the POSTSCRIPT language **grestore** operator.

In addition to the standard save and restore operators, the NeWS **currentstate** operator can be used to save the current graphics state as a NeWS graphics state object. The graphics state can be set to a given graphics state object with the

setstate operator. See Chapter 10, "NeWS Operator Extensions," for detailed descriptions of these operators. Note that a graphics state object cannot be accessed as a dictionary; it can only be saved and restored.

Creating a New Process

To create a new process, you use the **fork** operator:

procedure **fork** process

The **fork** operator creates a new process that is a child of the process that executes **fork**. The newly created child executes *procedure* in an environment that is a copy of its parent's environment. The **fork** operator does not start the child process running; the new process must wait its turn to run. The child process exits after executing *procedure*.

The next example forks a child process that executes a very simple procedure. The procedure performs the following tasks:

1. Prints the process information associated with the child process.
2. Prints the contents of the child process' operand stack.
3. Places the string (hello) on the child process' operand stack.
4. Prints the child's operand stack again.
5. Defines a key/value pair in the child's **userdict**.

```
{
    currentprocess ==
    currentprocess /OperandStack get ==
    (hello)
    currentprocess /OperandStack get ==
    /newkey 27 def
} fork pop
process (0x25d5dc,'Unnamed process', runnable, '==' )
[ 109 (mystring) /myname ]
[ 109 (mystring) /myname (hello) ]
```

In an interactive `psh` session, each line of your code is executed immediately when you press the [Return] key; the client connection process pauses and allows other processes to run while it waits for you to type each line. In this example, the child process runs immediately after you type fork pop and press [Return].

You can see that the child process has a different process identification number than its parent. Because the child was not given a name, its name defaults to `Unnamed process`. When the child prints its initial operand stack, you can see that the child inherited a copy of its parent's operand stack. The child then adds the string (hello) to the top of its operand stack.

If you type currentprocess == now, you will see that the parent is the current process again because the child exited after executing its procedure. If you then access the parent's dictionary stack for the new key defined by the child, you will see that the key is found in the parent's **userdict**. The child received a copy of

its parent's dictionary stack, so the child's changes to its dictionary stack are seen by the parent.

```
currentprocess ==
process (0x25c088, 'quakes client', runnable, '==')
newkey ==
27
```

The parent and child do not share operand stacks. When you print the parent's operand stack, you do not see the string (hello) that the child placed on its operand stack.

```
pstack
109 (mystring) /myname

quit
```

Process Scheduling: Allowing Other Processes to Run

Each NeWS process that has not yet exited can be either in a *runnable* state or a *blocked* state. A runnable process is ready to be run or is running. A blocked process is not ready to be run; a blocked process is waiting for some specified action to complete (for example, a blocked process might be waiting for another process to exit or an event to be delivered).

The server can only run one NeWS process at a time, so it keeps a list of all runnable processes and runs each one in turn (round-robin style scheduling). A process must be a cooperative client and periodically allow the next process to run, either by executing the **pause** operator (described in detail below) or by blocking. When a process executes the **pause** operator, the server gives all other runnable processes a chance to run; the process that executes **pause** is still runnable, and in fact, it will run immediately if no other runnable processes exist. If a process continues to run for 15 seconds without pausing or blocking, the server suspends the process and allows the next process to run.

A process may block in any of the following situations:

- The process is waiting for an event.

  If a process executes the **awaitevent** operator and no event is currently in the process' local event queue, the process blocks until an event is delivered (see Chapter 4, "Events," for details).

- The process is waiting for file I/O.

  The server may temporarily block a process while the process is reading from or writing to a file. Therefore, programmers should consider the possible synchronization problems when handling file I/O. See Section 3.3, "Using Monitors for Synchronization," for a description of how to use monitors to ensure proper synchronization.

- The process is waiting at a locked monitor.

  If a process attempts to lock a monitor that is already locked, the process blocks until the monitor is unlocked; see Section 3.3, "Using Monitors for Synchronization," for details.

- The process is waiting for another process to exit.

  When a process executes the **sleep** procedure or the **waitprocess** operator (both described below), the process blocks until another process exits.

- The process is suspended.

  A process can be blocked idefinitely with the **suspendprocess** operator or the **breakpoint** operator. A suspended process cannot run until it is explicitly unblocked with the **continueprocess** operator. See Section 3.1.8, "Suspending and Restarting Processes, " for details.

The following subsections describe **pause, waitprocess,** and **sleep.**

Pausing

You can use the **pause** operator to give all other runnable processes a chance to run:

– pause –

The **pause** operator causes the server to stop running the current process, giving all other runnable processes a chance to execute.

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

```
{
    5 {
        (child is running\n) print
        pause
    } repeat
} fork

5 {
    (parent is running\n) print
    pause
} repeat
```

This example forks a child process that prints the string child is running and then pauses. The child repeats this procedure 5 times. The parent also enters a loop that repeats 5 times. The parent prints the string parent is running and then pauses.

Type this example into a file and then give the filename as an argument to the `psh` command as follows:

```
% psh filename
```

When this example executes, the parent and child processes alternate between running and pausing. Thus the following output is printed to the screen:

```
% psh filename
child is running
parent is running
child is running
parent is running
child is running
parent is running
child is running
parent is running
child is running
parent is running
%
```

Waiting

You can use the **waitprocess** operator to block the current process until another process exits:

process **waitprocess** any

The **waitprocess** operator waits until *process* completes and then returns the value that was on top of *process*' operand stack at the time of completion. Until *process* completes, the process that executes **waitprocess** is not runnable.

The next example demonstrates the **waitprocess** operator. Edit your file so that it contains the following code:

```
{
    5 {
        (child is running\n) print
    } repeat
    (child is done)
} fork waitprocess pstack

5 {
    (parent is running\n) print
} repeat
```

Instead of using **pause**, this example uses the **waitprocess** operator after forking the child process. The **waitprocess** operator causes the child process to execute until it completes its procedure. Then the parent process executes again. When you run this example with `psh`, the following results are printed to the screen:

```
% psh filename
child is running
child is running
child is running
child is running
child is running
(child is done)
parent is running
parent is running
parent is running
parent is running
parent is running
%
```

Sleeping

You can use the **sleep** procedure to temporarily block a process; this procedure is provided by the POSTSCRIPT language files associated with the server (see Chapter 11, "Extensibility through NeWS Procedure Files"). The **sleep** procedure is described below:

num **sleep** –

Blocks the current process for *num* amount of time, where *num* is in units of 2^{16} milliseconds (65.36 seconds). Until the specified time has elapsed, the process is not runnable.

The **sleep** procedure is implemented with the **waitprocess** operator. The **sleep** procedure forks a process that exits after the specified amount of time; the **sleep** procedure executes a **waitprocess** on this forked process, causing the current process to block for the specified amount of time.

To demonstrate the **sleep** procedure, edit the code file you used in the previous two examples. Remove the **waitprocess** operator and add the **sleep** procedure as follows:

```
{
    5 {
        (child is running\n) print
        .01 sleep
    } repeat
} fork

5 {
    (parent is running\n) print
    .01 sleep
} repeat
```

When you run the above example, you will get the same results as you did with **pause**, but more time will elapse between the printing of each line.

Examining the Process Execution State

Each process has a read-only **State** key that indicates its current execution state. The **State** key can have one of the following values:

runnable The process is running or is scheduled to be run.

dead The process is dead; the process has exited and no references to it remain. (Note that this value will never be seen by the user.)

zombie The process has exited, but other processes still have references to it.

input_wait The process is waiting for an event.

IO_wait The process is waiting for file input/output.

mon_wait The process is waiting at a monitor.

proc_wait The process is waiting for another process to exit.

breakpoint The process is suspended, normally for debugging.

Note that the last five **State** values listed above represent various types of blocked processes.

You can determine a process' execution state by examining the value of its **State** key. The following example forks a process that prints its **State** value and then exits. While the child is running, its **State** is **runnable**. After the child process exits, it is left on its parent's operand stack. Because it is still referenced but has exited, the child's state is then **zombie**.

```
% psh
executive
Welcome to X11/NeWS Version 2.1

{ currentprocess /State get == } fork
/runnable
pstack
process (0x309498,'Unnamed process', zombie, null)

quit
```

NOTE *In an interactive* `psh` *session, you do not need the* **pause** *or* **waitprocess** *operator to allow a child process to run; each line of your code is executed immediately when you press the* Return *key, and there is an implicit pause when the server is waiting for you to type.*

Destroying Processes

When a process finishes executing its procedure, it exits. If no references to it exist, its state is **dead**, and it is garbage collected. If references do exist, its state is **zombie**, and the server sends a **ProcessDied** event to each process that has expressed interest (see the subsection "ProcessDied Events" in Section 4.6, "System-Generated Events," of Chapter 4, "Events").

You can kill a runnable process with the **killprocess** operator:

process **killprocess** –

Sends a `killprocesserror` error to the specified *process*. If that *process* is not able to catch or handle that error, the *process* exits. (The POSTSCRIPT language **stopped** operator can be used to catch all errors encountered during a given piece of POSTSCRIPT language code. The **errordict** dictionary can contain code to handle any run time error. See Section 3.4, "Handling Errors," for more information.)

The following example demonstrates the **killprocess** operator. The example forks a child process and lets the child run while the parent sleeps. The child loops, printing the string hello to the screen. When the parent finishes its sleep, it prints the child's **State**, which is **runnable**. The parent then kills the child process and prints the child's **State** again. Because the parent still has a reference to the child process, the child's **State** is **zombie**. To remove the reference, the parent undefines the child's name from its **userdict**; the child process can then be garbage collected by the server. You can type this example into a file and run it with `psh`.

```
/ChildProcess {
    { (hello\n) print pause } loop
} fork def

0.001 sleep
ChildProcess /State get ==

ChildProcess killprocess
pause
ChildProcess /State get ==

userdict /ChildProcess undef
```

When you run this example, the following lines are printed to the screen:

```
% psh  filename
hello
hello
.
.
.
/runnable
/zombie
```

You will often want to kill a whole process group at once; see Section 3.2, "Creating and Manipulating Process Groups," for details.

Suspending and Restarting Processes

In addition to temporarily pausing, waiting, or sleeping, a process can indefinitely suspend itself or another process. The **suspendprocess** operator suspends a specified process:

process **suspendprocess** –

This operator suspends *process*. The *process* will not run again until another process executes **continueprocess** on it.

A process can restart a suspended process with the **continueprocess** operator:

process **continueprocess** –

This operator restarts *process*, which had been suspended.

A process might want to suspend itself after asking another process for information; the other process would restart the suspended process after sending the requested information.

The **breakpoint** operator can be used to suspend the current process:

– **breakpoint** –

This operator suspends the current process. It is usually used for debugging.

The following example demonstrates the **suspendprocess** and **continueprocess** operators. This example creates a canvas and a togglecolor procedure that alternates painting the canvas black and white. The example forks a child process that executes the togglecolor procedure. When you type this example in an interactive `psh` session, the child process runs, causing the canvas to flash. When you suspend the child by typing ChildProcess suspendprocess, the canvas stops flashing. You can restart the child by typing ChildProcess continueprocess.

```
% psh
executive
Welcome to X11/NeWS Version 2.1

/MyCanvas framebuffer newcanvas def
0 0 250 250 rectpath
MyCanvas reshapecanvas
MyCanvas /Mapped true put
MyCanvas setcanvas
0 fillcanvas

/color 0 def

/togglecolor {
    {
        /color 1 color sub def
        color fillcanvas
        0.01 sleep
    } loop
} def

/ChildProcess { togglecolor } fork def      % Fork child.
                                            % Canvas flashes black/white.
ChildProcess suspendprocess                 % Suspend child.
                                            % Canvas stops flashing.
ChildProcess continueprocess                % Continue child.
                                            % Canvas flashes again.
ChildProcess killprocess                    % Kill child process.

quit                                        % Quit psh.
```

Using the psps Utility

You can use the psps utility to print information about all the processes currently in the server. Just type **psps** to a shell prompt; the utility prints eight items of information for each process, including the process ID, state, priority, name, and size of the execution, operand, and dictionary stacks. See the manual page for psps in the *X11/NeWS Server Guide* for a more detailed description of the information printed by psps.

3.2. Creating and Manipulating Process Groups

As discussed previously, NeWS processes exist in a process tree. Within the tree, the processes are grouped into *process groups*. When a child process is forked, it is placed in its parent's process group. A process can be removed from its process group and placed in its own, new process group with the following operator:

process **newprocessgroup** –

Removes *process* from its process group, and puts *process* in a new process group of its own. Children forked by *process* will be in the new process group.

You can inspect the processes in a process group with the **getprocessgroup** operator:

process **getprocessgroup** array
null **getprocessgroup** array

If *process* is specified as the argument, **getprocessgroup** returns an array of all the processes in the process group of *process*; if **null** is specified as the argument, **getprocessgroup** returns an array of all the processes in the current process's process group. If *process* is a zombie process, it is the only process in the array because zombie processes are not associated with any process group. (If a process becomes a zombie, it is removed from its process group.)

The following example demonstrates the **newprocessgroup** and **getprocessgroup** operators:

```
{
    { (hello! \n) print } fork pop
    { (goodbye! \n) print } fork
    getprocessgroup ==
    pause
} fork newprocessgroup

pause
```

This example forks a child process and places it in a new process group. The child process forks two children of its own; these two children are in the same process group as their parent. The **getprocessgroup** operator is used to return the processes in the newly formed process group.

When you type this example into a file and run it with `psh`, the following results are printed to the screen:

```
% psh filename
[ process (0x30d398, 'Unnamed process', runnable, null)
  process (0x30d020, 'Unnamed process', runnable, null)
  process (0x30cc88, 'Unnamed process', runnable, '==')]
hello!
goodbye!
```

One of the main advantages of using process groups is that you can kill all the processes in one process group with the following operator:

process **killprocessgroup** –

This operator kills *process* and all other processes in the same process group.

The **getprocesses** operator can be used to return an array of all processes currently in the server:

– getprocesses array

This operator returns an array of process groups and zombie processes. Each process group is returned as an array that contains all the processes in the process group. Each zombie process is returned as an array containing only itself.

The following example demonstrates the **killprocessgroup** and **getprocesses** operators:

```
{
    {
        { (hello! \n) print pause } loop
    } fork pop
    {
        { (goodbye! \n) print pause } loop
    } fork pop
pause
} fork dup newprocessgroup

0.001 sleep
getprocesses ==
killprocessgroup
pause
(processgroup killed \n) print
getprocesses ==
```

This example is similar to the previous example, except that the two grandchild processes both enter loops. Instead of waiting until the grandchild processes exit their procedures, the **killprocess** operator is used to kill all the process in the newly formed process group. The **getprocesses** operator is used to show the processes that exist before and after the **killprocess** operator is executed.

When you type this example into a file and give the filename as an argument to the `psh` command, the following results are printed to the screen:

```
% psh filename
hello!
goodbye!
hello!
goodbye!
 .
 .
 .
[ . . . .]
processgroup killed
[ . . . .]
```

The arrays returned by **getprocesses** are not enumerated in the box above; the contents of the arrays will vary depending on the applications and tools you are running when you execute this example (the **getprocesses** operator returns all the

processes currently running in the server). You will notice that the process group you created in this example is listed by **getprocesses** before, but not after, you kill the process group with **killprocessgroup**.

Note that when a connection to the server is established, the server places the client's connection process in its own process group. When the connection process reaches the end of file on the connection, its process group is killed with currentprocess killprocessgroup.

3.3. Using Monitors for Synchronization

When two processes access the same data structure, you must ensure proper synchronization. For example, if two processes are writing to the same file, you must ensure that one process completes its changes to the file before the other process begins to make its changes to the file. Synchronization problems may occur because the server may block a process at any time during a file I/O operation; the changes made by the first process could still be in progress when the server blocks the process, allowing the second process to run.

The following example illustrates this situation. Two processes are forked, and both processes print strings to the standard output file. One process prints ones; the other process prints zeroes. Neither process pauses, so you might expect one process to run to completion before the other process starts. However, the server periodically blocks these processes while they are writing to the file, causing ones and zeroes to be mixed in the output that is printed to the standard output file.

```
{
   5000 { (1) print } repeat
} fork

{
   5000 { (0) print } repeat
} fork

.01 sleep
```

When you run this example, the output looks something like the following:

```
% psh filename
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
etc.
```

To allow the first process to finish printing all its ones before the second process starts printing zeroes, you must use a *monitor*. A monitor is a NeWS object that can be in one of two states: locked or unlocked. A monitor can be accessed by only one process at a time; thus, monitors can be used to implement synchronization.

You create a monitor with the following operator:

– **createmonitor** monitor

The **createmonitor** operator creates a new monitor object and leaves it on the operand stack.

To use a monitor, you execute the following operator:

monitor procedure **monitor** –

The **monitor** operator executes *procedure* with *monitor* locked. At any given time, only one process may have *monitor* locked. If a process attempts to lock a monitor that is already locked, the process blocks until the monitor is unlocked.

You can use the **monitorlocked** operator to inspect the state of a monitor object:

monitor **monitorlocked** boolean

This operator returns **true** if *monitor* is locked; otherwise, it returns **false**.

The following example demonstrates the use of monitors. The code is the same as the previous example, except that each forked process locks a monitor before entering the loop that prints characters to the screen.

```
/mymonitor createmonitor def

{
   mymonitor {
      5000 { (1) print } repeat
   } monitor
} fork

{
   mymonitor {
      5000 { (0) print } repeat
   } monitor
} fork

.01 sleep
```

When you run this example with `psh`, the following output is printed to the screen:

```
% psh filename
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
111111111111111111111111111111111111111111
etc.
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
000000000000000000000000000000000000000000
etc.
```

This time, all the ones are printed first, followed by all the zeroes. The first process locks a monitor before starting to print ones. When the server blocks the first process, the second process tries to lock the monitor so that it can execute its procedure (printing zeroes), but the monitor is already locked. Thus, the second process blocks. The server gives the first process a chance to run again. Until the first process unlocks the monitor by finishing printing its ones, the second process cannot run. After the first process finishes and the monitor is unlocked, the second process locks the monitor and prints zeroes. Thus, the ones and zeroes are not mixed in the resulting output.

This example has important implications for client applications. Whenever two or more processes access a shared data structure, monitors should be used. For example, monitors are necessary when writing results from the server to the client's input connection file (see Chapter 6, "C Client Interface," for details).

3.4. Handling Errors

Error handling in the X11/NeWS server is much like standard POSTSCRIPT language error handling, except that each process has its own, private **errordict** and **$error** located in **systemdict**. Accessing the **errordict** and **$error** keys in **systemdict** is equivalent to accessing these same keys in the current process. Each process dictionary has the following read/write keys:

- **errordict**

  The value of this key is a dictionary that maps each type of error the process might receive to an error handler (a procedure). Each client connection process receives a copy of the NeWS listener's **errordict**, which by default maps each error that the server can generate to the **defaulterroraction** operator (described below). When a child process is forked, it shares its parent's **errordict**.

- **$error**

  The value of this key is a dictionary that contains information about the last error that the process encountered. The dictionary is filled by the **defaulterroraction** operator when an error occurs. The **$error** dictionary is similar to the POSTSCRIPT language **$error** dictionary, but it has one additional key named **message**; if the value of the process' **ErrorDetailLevel** key is greater than zero, **message** contains a string that describes the context of the error. If the **defaulterroraction** operator has not been executed, the value of **$error** is **null**.

The **defaulterroraction** operator is described below.

any errorname **defaulterroraction** –

This operator produces an **$error** dictionary for the current process as if the error specified by *errorname* had been encountered while executing the object *any*. The operator then executes the POSTSCRIPT language **stop** operator.

You can control the amount of information written in the **$error** dictionary with a process' **ErrorDetailLevel** key:

- **ErrorDetailLevel**

  This key's value is an integer that controls the amount of detail that is included in the default error handler's error report. Setting **ErrorDetailLevel** to 0 (the default) produces a minimum of error reporting. Setting it to 1 records a more descriptive message in the **$error** dictionary, and setting it to 2 records the contents of the dictionary, execution, and operand stacks.

Each process also has an **ErrorCode** key that contains the current error code of the process. The error code can be any of the standard POSTSCRIPT language error codes or one of the NeWS error codes. See Chapter 9, "NeWS Type Extensions," for a list of all the error codes and a description of the NeWS error codes.

3.5. Controlling Dictionary Sharing Between Parent and Child Processes

When you fork a child process, the child inherits its parent's dictionary stack. Most of the time, this default behavior is desirable. However, you may occasionally want to prevent a child process from inheriting its parent's **userdict**. The following code fragment demonstrates one way in which you can keep the parent's **userdict** private:

```
systemdict /myuserdict userdict put     % Save userdict as myuserdict.
end                                     % Pop it off of dict stack.
growabledict begin                      % Push a new dict on stack.
{
        .                               % Fork child process that now has
        .                               % the new dictionary as its
        .                               % userdict.
} fork

end                                     % Pop off child's userdict.
myuserdict begin                        % Push parent's userdict back on stack.
systemdict /myuserdict undef            % Remove reference to myuserdict.
```

Note that this example uses the **growabledict** procedure, which is provided by the POSTSCRIPT language extensibility files loaded when the server is initialized. The **growabledict** procedure creates a large, growable dictionary and leaves it on the operand stack.

4

Events

An *event* is an object that represents a message between NeWS processes. An event can be generated by the server or by any NeWS process. Events that originate from the server are known as *system-generated* events; events that originate from NeWS processes are known as *process-generated* events.

An event can be delivered to any NeWS process. Events can transmit any kind of information and thus serve as a general interprocess communication mechanism. Some system-generated events report user manipulation of input devices and are therefore known as *input events*.

An event is implemented as a NeWS type extension that can be accessed as a dictionary. A NeWS process can create an event object with the **createevent** operator. The newly created event dictionary contains keys with system-supplied names and initial values of null or zero. The process can then give the desired values to the keys and send the event into distribution.

A process sends an event into the server's distribution mechanism with the **sendevent** operator. System-generated events are automatically sent into distribution immediately after they are generated. The server's distribution mechanism accumulates events in a *global event queue* and distributes a copy of each event to NeWS processes that are interested in receiving the event.

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

This chapter describes NeWS events, event operations, and the server's event distribution mechanism.

The eventtype Extension

Each event is an object of type **eventtype**, which is a NeWS extension to the POSTSCRIPT language. Each **eventtype** object can be accessed as a POSTSCRIPT language dictionary. An event dictionary contains keys that describe the following properties (the keys are listed in parentheses):

- The identity of the event (**Action, Name, Serial, Process**)
- The location or destination of the event (**Coordinates, XLocation, YLocation, Canvas**)

- The time after which the event can be distributed (**TimeStamp, TimeStampMS**)
- Whether the event is in the server's global event queue (**IsQueued**)
- The interest that matched the event (**Interest**)
- The characteristics of an interest event (**Exclusivity, IsInterest, IsPreChild, Priority, Synchronous**)
- The keyboard keys and mouse buttons that were down at the time the event was generated (**KeyState**)
- Additional client-specific information included in the event (**ClientData**)

Most of the event keys are discussed in this chapter; a description of each key is provided in Chapter 9, "NeWS Type Extensions."

Event Operators

The NeWS language includes a variety of operator extensions to be used on events. The event operators provide the following functionality (the operator names are listed in parentheses):

- Creating events (**createevent**)
- Distributing events (**sendevent, deliverevent**)
- Enabling and disabling reception of events (**expressinterest, revokeinterest**)
- Retrieving an event from the process' local event queue (**awaitevent**)
- Manipulating the event distribution mechanism (**blockinputqueue, recallevent, redistributeevent, unblockinputqueue**)
- Counting the number of events in a process' local event queue (**countinputqueue**)
- Returning the keystate, time, and coordinates of the last event distributed from the global event queue (**lasteventkeystate, lasteventtime, lasteventx, lasteventy**)
- Creating canvas crossing events (**postcrossings**)
- Creating and returning an event-logger process (**seteventlogger, geteventlogger**)
- Setting and inspecting the event synchronization state of a process (**setcompatinputdist, getcompatinputdist**)

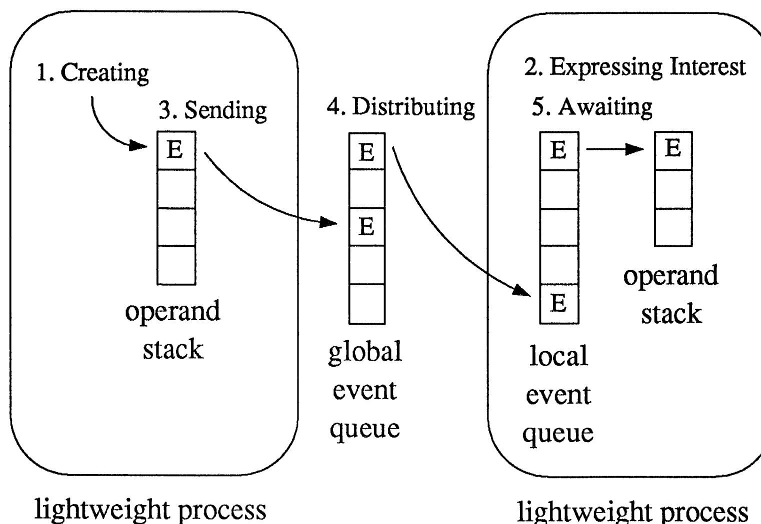
Most of the event operators are described in this chapter. A description of all NeWS operators is provided in Chapter 10, "NeWS Operator Extensions."

4.1. Overview of Event Distribution

The distribution of an event consists of five basic steps. The following figure is a schematic representation of these five event distribution steps. The figure shows one lightweight process sending an event and another lightweight process receiving it. Each of these five steps occurs at a separate instant in time, but all five steps are shown on the same diagram for compactness. Therefore, all six E's

shown in the diagram represent the same event at different times.

Figure 4-1 *The five steps in an event's distribution*



The five steps in an event's distribution are described below.

1. Creating the event.

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

2. Expressing interest in the event.

   Before a process can receive an event, the process must express interest in that type of event with the **expressinterest** operator.

3. Sending the event to the server's global event queue.

   A process sends an event to the global event queue with the **sendevent** operator. System-generated events are automatically sent to the global event queue after the server creates them. Sending an event to the server's global event queue is commonly referred to as *sending the event into distribution* or *sending the event into the event distribution mechanism.*

   When an event is sent to the global queue, it is sorted into the queue according to the value of its **TimeStamp** key; events with smaller timestamp values are placed closer to the head of the queue. When the server generates an event, the current time is stored in the event's **TimeStamp** key. Other events have whatever **TimeStamp** value is specified by the process that creates them. An event is never distributed before the time indicated in its **TimeStamp** key. Therefore, processes can specify that an event be distributed at some time in the future. For more information about setting an event's **TimeStamp**, see the subsection "Specifying the Time of an Event's Distribution" in the next section.

   In the figure above, the event represented by E is sorted into the middle of the global event queue (step 3). Then the events above E are distributed,

leaving E at the head of the global event queue. When the current time is equal to or larger than E's **TimeStamp**, the server distributes E to interested processes (step 4).

4. Distributing the event to the local event queues of interested processes.

   Distribution of an event is initiated whenever the event at the head of the global event queue has a **TimeStamp** that is less than or equal to the server's current time. When this occurs, the event is removed from the queue and is compared with the interests to locate matches. An event is not necessarily compared to all the interests; the value of the event's **Canvas** key determines which interests are compared to the event. (The search procedure is described in detail in Section 4.5, "Event Distribution: Matching an Event to Multiple Interests.")

   When an event is compared to an interest, the server attempts to match four of the dictionary keys in the event to the same four keys in the interest: the **Name**, **Action**, **Process**, and **Serial** keys must match according to specific rules before an interest is said to match an event. (The matching rules are given in Section 4.3, "Rules for Matching Events to Interests.")

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

   This distribution procedure allows the server to distribute a single event to many processes that are interested in that event. If a matching interest has its **Synchronous** key set to **true** or the process that has a matching interest has executed the **setcompatinputdist** operator to set its event synchronization mode to **true**, the process is given a chance to run before the next event is removed from the global event queue. Otherwise, the server continues to distribute events.

   In the figure above, the server finds that the second process has a matching interest for event E; therefore, the server distributes a copy of E to the local event queue of the second process. The figure depicts E being placed at the bottom of the process' local event queue. After the process retrieves any events that are ahead of E in the queue, E is then at the head of the queue, ready to be retrieved.

5. Retrieving the event from the local event queues of processes with matching interests.

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

If a process receives an event that has matched a **Synchronous** interest, the process is responsible for unblocking the global event queue with the **unblockinputqueue** operator. See Section 4.7, “Synchronizing Input with Multiple Processes,” for more information about event synchronization.

4.2. Basic Event Operations

This section discusses basic event operations such as creating and sending events, expressing and awaiting interests, setting and inspecting an event's location, recalling events, and revoking interests.

You can use the psh command to run the examples in this chapter. The first two examples in this section must be typed in an interactive psh session. Be sure to type all the code sequentially from the start to the quit of each interactive psh session. To run most of the other examples in this chapter, you can type the example into a file and then give the filename as an argument to the psh command as follows:

```
myprompt% psh filename
```

See the psh manual page in the *X11/NeWS Server Guide* for more information about psh.

Creating an Event

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

– **createevent** event

This operator creates an event object and places it on the top of the operand stack for the current process. The event keys with non-numeric values are initialized to **null**, and the event keys with numeric values are initialized to zero.

The following example creates an event and specifies values for its **Name** and **Action** fields. Each of these fields can take any type of NeWS object as its value. Here, each value is specified as a string.

```
myprompt% psh
executive
Welcome to X11/NeWS Version 2.1

createevent                       % Create an event.
dup begin                         % Put event on dictionary stack.
   /Name (Hello) def              % Set Name key value.
   /Action (There!) def           % Set Action key value.
end                               % Remove event from dictionary stack.
                                  % Note event is still left on operand stack.
```

An event's **Name** and **Action** fields are two of the four fields that the server uses to match the event to interests; for an interest to match the example event created here, the interest's **Name** and **Action** must contain the same strings as the event.

Expressing Interests

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

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

This operator allows all events that match the *event* argument to be received by the specified *process* or by the current process if no *process* argument is specified. The *event* is known as an interest. The process for which the interest is expressed is stored in the interest's **Process** key. The interest is added to the array of interests stored in the process' **Interests** key. If *event* is already an active interest, the call to **expressinterest** is ignored.

An interest's type is still **eventtype**. Interests can be distinguished from other events by the **IsInterest** key; when an event is expressed as an interest, the server sets the event's **IsInterest** key to **true**.

Copying an Event Before Expressing Interest

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

If you want to send an event into distribution and also express interest in it, you can perform the following steps:

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

To make a copy of an event, you can create a new, empty event and then use the POSTSCRIPT language **copy** operator as follows:

```
dup                 % Duplicate the reference to the event on the stack
                    % of the operand stack in the previous example.
createevent         % Create an empty event.
copy                % Copy the duplicate event into the newly
                    % created event.
expressinterest     % Express interest in the copy. Note that the
                    % original event is still left on the operand stack
                    % so that it can be sent into distribution
                    % in the next example.
```

Changing and Reusing Interests

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

However, none of the other key values can be changed once an interest has been expressed. If you attempt to change another key's value, an `invalidaccess`

error is generated and all the key values remain the same. To change any of the other key values, you must revoke the interest (using the **revokeinterest** operator) and then change and re-express the interest. See the subsection "Recalling Events and Revoking Interests" for details.

Sending an Event into Distribution

After an event has been created, it can be sent into the server's event distribution mechanism with the **sendevent** operator:

event **sendevent** –

This operator sends *event* into the server's distribution mechanism; it places *event* in the server's global event queue to be distributed to interested processes. The *event* is sorted into the global event queue according to the value of its **TimeStamp** key, which should be given a value by the process that creates the *event.*

The server removes an event from the head of the global event queue when the event's **TimeStamp** value is less than or equal to the server's current time. When an event is removed, it is compared with interests to locate matches. When a match is found, a copy of the event is distributed to the process that has the matching interest; the copy is placed on the local event queue of the process.

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

```
sendevent        % Send into distribution the event that was left
                 % on the operand stack in the previous example.
                 % The event matches the interest previously
                 % expressed by this process, so a copy of the
                 % event is distributed to this process' local
                 % event queue.
```

Awaiting Events

To retrieve the events from a process' local event queue, you use the **awaitevent** operator:

– **awaitevent** event

Returns *event* from the process' local event queue. When no event is contained in the process' local event queue, this operator causes the process to block; when an event arrives in the local event queue, **awaitevent** places the event on top of the process' operand stack and unblocks the process. If an event is waiting on the local queue when **awaitevent** is called, the event is immediately placed on the process' operand stack.

The following example executes **awaitevent** to retrieve the event that was placed in the process' local event queue in the previous example. This example then prints the values of the event's **Name** and **Action** keys.

```
awaitevent              % Retrieve event from local event queue.
dup /Action get         % Get Action.
exch /Name get          % Get Name.
= =                     % Print top two items from operand stack.
Hello
There!

quit                    % Quit psh.
```

Using Arrays in an Interest's Name, Action, or Canvas Key

To allow an interest to match more than one type of event, you can use an array of values for the interest's **Name**, **Action**, or **Canvas** key. An array in one of these keys allows the interest to match an event that has any of the array values in the event's corresponding key.

For example, suppose the **Name** key of an interest has the following array as its value:

[(Hello) (GoodBye)]

This interest can match an event that has either the string (Hello) or (GoodBye) as the value of its **Name** key.

The following example creates an interest with an array in its **Name** key, expresses the interest, and then sends two matching events; each matching event has a different value in its **Name** key. The events are then retrieved from the local event queue, and their **Name** values are printed to the screen. Try typing this example into an interactive `psh` session.

```
% psh
executive
Welcome to X11/NeWS Version 2.1

createevent dup                              % Create the interest.
/Name [ (Hello) (GoodBye) ] put
expressinterest

createevent dup                              % Send one matching event.
/Name (Hello) put
sendevent

createevent dup                              % Send another matching event.
/Name (GoodBye) put
sendevent

awaitevent                                   % Retrieve first event from
/Name get =                                  % local event queue.
Hello

awaitevent                                   % Retrieve second event from
/Name get =                                  % local event queue.
GoodBye

quit                                         % Quit psh.
```

Setting and Inspecting an Event's Location

The **eventtype** dictionary contains **XLocation** and **YLocation** keys, which hold the *x* and *y* coordinates, respectively, at which the event occurred. The **eventtype** dictionary also contains a **Coordinates** key, which holds an array of length two; the first array element is the event's *x* coordinate, and the second array element is the event's *y* coordinate. Although it is not usually necessary, a process can set these keys before sending an event into distribution. The server sets these keys before sending most system-generated events; the server sets an event's coordinates to the location of the mouse pointer at the time the event is generated. When the event coordinates are retrieved, they are given with respect to the current coordinate system. (Thus, if an event's coordinates are retrieved both before and after changing the current coordinate system, the coordinate numbers will be different, but they will correspond to the same position on the screen.)

The following example demonstrates the use of the **XLocation** and **YLocation** keys. This example uses system-generated mouse button events. When a mouse button is pressed, the server generates an event that has the value of its **Name** key set to **/LeftMouseButton**, **/MiddleMouseButton**, or **/RightMouseButton**, depending on which mouse button is pressed; the value of the **Action** key is set to **/DownTransition**. When the mouse button is released, another event is generated with the same **Name** and with an **Action** of **/UpTransition**. Mouse events are described in detail in Section 4.6, "System-Generated Events."

This example creates and maps a canvas, and then it expresses interest in left and right mouse button presses on that canvas. The interest has an array in its **Name**

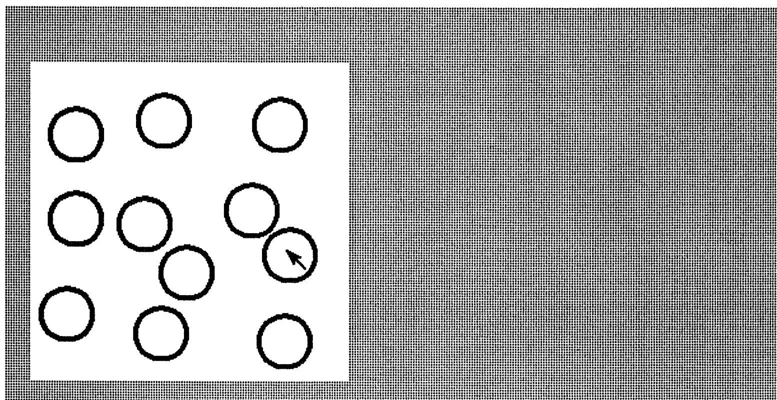
field; the array contains the names **/LeftMouseButton** and **/RightMouseButton**, allowing this interest to match either left or right button events.

After expressing interest in left and right button presses, the example enters an **awaitevent** loop. When a left button press event is returned, the *x* and *y* coordinates of the event are retrieved. The example then draws a circle centered about the (*x*, *y*) location of the event. The loop is exited when the right mouse button is pressed.

```
/MyCanvas framebuffer newcanvas def                  % Create MyCanvas.
25 25 translate
0 0 250 250 rectpath
MyCanvas reshapecanvas
MyCanvas /Mapped true put
MyCanvas setcanvas
1 fillcanvas

createevent dup begin                                 % Create interest.
    /Name [/LeftMouseButton /RightMouseButton] def
    /Action /DownTransition def
    /Canvas MyCanvas def
end expressinterest                                   % Express interest.
                                                      % Print instructions.
(Press the left mouse button over MyCanvas to draw some \n) print
(circles.  Press the right mouse button to quit the loop. \n) print

4 setlinewidth 0 setgray
{
    awaitevent begin                                  % Retrieve event.
        Name dup
        /LeftMouseButton eq {                         % If left press,
            XLocation YLocation 20 0 360 arc stroke   % draw circle.
        } if
    end                                               % If right press,
    /RightMouseButton eq {exit} if                    % exit.
} loop
```

Run this example with `psh` and draw some circles by clicking the left mouse button over the canvas. The following figure illustrates an example of some circles drawn in this way:

Figure 4-2 *Circles drawn in the canvas*



Specifying the Time of an Event's Distribution

The **eventtype** dictionary contains a **TimeStamp** key, whose value indicates the time after which the event may be removed from the global event queue for comparison with interests. When an event is sent to the global event queue, it is sorted into the queue according to its **TimeStamp** value; events with smaller timestamps are placed closer to the head of the queue. An event cannot be removed from the global event queue before the current time is equal to the time specified by the event's **TimeStamp** value. Thus, when an event contains a **TimeStamp** value that specifies a time in the future, the event must remain in the global event queue until the specified time is reached. The **TimeStamp** key's value is measured in units of 2^{16} milliseconds (65.5 seconds).

The server also offers another interface to an events's timestamp: the **eventtype** dictionary's **TimeStampMS** key. This key's value is similar to the **TimeStamp** key, except that it is given in units of milliseconds instead of 2^{16} milliseconds and is an integer instead of a real number. The **TimeStampMS** key is useful for precise integer arithmetic with event timing.

Before you send an event into distribution, you can set the value of either the **TimeStamp** key or the **TimeStampMS** key. The following example demonstrates how the **TimeStamp** value can be used:

```
/MyCanvas framebuffer newcanvas def          % Create MyCanvas.
25 25 translate
0 0 250 250 rectpath
MyCanvas reshapecanvas
MyCanvas /Mapped true put
MyCanvas setcanvas
1 fillcanvas

createevent dup begin                        % Create interest in timer events
  /Name [ /Timer /RightMouseButton ] def     % and right button presses.
  /Canvas MyCanvas def
end expressinterest                          % Express interest.

/delay .007 def                              % Initialize variables.
/color 0 def

(Watch the canvas change color. \n) print    % Print instructions.
(Click right mouse button to quit. \n) print

{
  createevent dup begin                      % Create timer event.
    /Name /Timer def
    /Canvas MyCanvas def
    /TimeStamp currenttime delay add def
  end sendevent                              % Send timer event.
  awaitevent /Name get                       % Retrieve event.
  /RightMouseButton eq { exit } if           % If right button event, exit.
  color fillcanvas                           % If timer event, paint MyCanvas.
  /color 1 color sub def                     % Toggle color.
} loop
```

This example creates and maps a canvas, and then it expresses interest in timer events and right mouse button events for that canvas. Two variables are initialized: a delay variable is initialized to approximately one-half of a second (.007 in timestamp units), and a color variable is initialized to 0 (black).

The main loop starts by sending a timer event with a **TimeStamp** equal to the current time plus the delay. The code then executes **awaitevent**. When an event is returned, the value of its **Name** is placed on the operand stack. If the event is a right mouse button event, the **exit** operator is executed. If the event is a timer event, the previously created canvas is painted with color. The color is then toggled between black and white. While the loop executes, the canvas changes color from white to black to white to black.

Run the example and watch the canvas change color. Then edit the code file to change the value of the delay variable to be .014 instead of .007. Run the example again and watch the canvas change color at a slower rate. Each timer event is released from the global event queue when the current time equals its **TimeStamp**, so the canvas changes color more slowly after the delay is increased.

Specifying Additional Event Information

The **eventtype** dictionary contains a **ClientData** key that can hold any type of NeWS object. This key is useful if you need only one additional key in the event dictionary. Although new keys can be added to any NeWS magic dictionary, the addition of the first new key uses a significant amount of memory. If you only need to use one additional key, this memory cost can be avoided by using the **ClientData** key.

Recalling Events and Revoking Interests

To recall an event from distribution, you can use the **recallevent** operator:

event **recallevent** –

Removes *event* from the server's global event queue to prevent it from being distributed. This operator is only effective if *event* has not yet been distributed. This operator is useful when you are waiting for several mutually exclusive events; when the first event occurs, you can immediately recall the other events. For example, you might use **recallevent** to recall a timer event.

To revoke an interest, you can use the **revokeinterest** operator:

event **revokeinterest** –
event process **revokeinterest** –

Revokes interest in *event*, where *event* is an interest that has been previously expressed. The optional *process* argument specifies the process on whose behalf the interest is revoked; if no process is specified, interest is revoked on behalf of the current process. If you specify a *process* argument that is not the same as the value of the interest's **Process** key, you will receive an `invalidaccess` error. Likewise, if you specify no *process* argument, but the current process is not the same as the **Process** value of the interest, you will receive an `invalidaccess` error.

4.3. Rules for Matching Events to Interests

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

Rules for Matching Name And Action Key Values

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

- If the interest's key value is **null**, it matches anything in the key of the event.
- If the interest's key value is an array, at least one of the array's elements must be identical to the event's key value. If the interest's key value is a dictionary, at least one of the dictionary's keys must be identical to the event's key value.
- If the interest's key value is anything other than an array, a dictionary, or **null**, it must be identical to the event's key value.

- If the interest's key value is the name **AnyValue** or is an array or dictionary that contains **AnyValue**, it matches anything in the key of the event. If a dictionary contains both **AnyValue** and a value identical to the event's key value, the identical value is used as the match. (Note that **AnyValue** can be used as the name of a dictionary key, whereas **null** cannot be used as a key name.)
- If the event's key value is **null** or **AnyValue**, it matches only **null** or **AnyValue** in the corresponding key of the interest.

Rules for Matching Process Key Values

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

- If the event's **Process** key value is **null**, it matches anything in the **Process** key of the interest. Thus, an event with **null** in its **Process** key can be delivered to any process that has a matching interest.
- If the event's **Process** key value is a specific process, this value must be identical to the value of the interest's **Process** key. Thus, an event with a process specified in its **Process** key may only be delivered to that specified process.

Rules for Matching Serial Key Values

The value of an event's **Serial** key, which is read-only, is used to indicate the order in which events are removed from the global event queue. When an event is removed from the head of the global event queue, the value of its **Serial** key is set to a numeric value given by a monotonically increasing counter (the counter is incremented each time an event is removed from the global queue). If the event is then successfully matched with an interest, the interest's **Serial** key is automatically set to the value that the event's **Serial** key contains. The server allows an event to match an interest only if the interest's serial number is less than that of the event; this restriction prevents an event passed to the **redistributeevent** operator from repeatedly matching the same interests.

For a description of the **redistributeevent** operator, see the subsection "Exclusive Interests" in Section 4.5, "Event Distribution: Matching an Event to Multiple Interests."

4.4. Post-Match Processing: Specifying a Dictionary for an Interest's Name, Action, or Canvas Key

When an interest has a dictionary as the value of its **Name, Action,** or **Canvas** key, the server performs some post-match processing when a matching event is found; the type of post-match processing depends on whether the event's key value matches a key with an executable or non-executable value in the interest's dictionary. These two cases are described in the next two subsections.

Specifying Non-Executable Dictionary Values

If the dictionary value associated with the interest's matching key is non-executable, the value is stored in the corresponding field of the event copy (that is, in the **Name, Action,** or **Canvas** field). The copy of the event is then placed in the local event queue of the process that had the matching interest. When the event is returned with **awaitevent**, the newly substituted key value can be retrieved.

This post-match behavior for non-executable dictionary values is demonstrated by the following example:

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

createevent dup begin                          % Create an event.
   /Name 3 dict dup begin                      % Create dict for the Name field.
      /LeftMouseButton (Left Button Went Up) def
      /MiddleMouseButton (Middle Button Went Up) def
      /RightMouseButton (Right Button Went Up) def
   end def
   /Action /UpTransition def                   % Make Action be button releases.
   /Exclusivity true def
end expressinterest                            % Express interest in the event.
                                               % Print instructions.
(Try pressing the left and middle mouse buttons. \n) print
(Then press the right mouse button to exit. \n) print

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

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

The **Exclusivity** key of each interest is set to true so that the interests are *exclusive*; an event that matches an exclusive interest is not compared to any other interests. For more information about exclusive interests, see the subsection "Exclusive Interests" in Section 4.5, "Event Distribution: Matching an Event to Multiple Interests."

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

Went Up), the loop is exited.

Run this example with psh and then press the left and middle mouse buttons. Each time you press or release a mouse button, a message is printed to the screen in your psh session. To exit the loop, press and release the right mouse button.

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

Specifying Executable Dictionary Values

If the dictionary value associated with the interest's matching key is executable, the corresponding event field is not modified; instead, the executable dictionary value is executed immediately after the received event is placed on the top of the process' operand stack by **awaitevent**. If more than one of the fields have executable values in their dictionaries, the **Name** value is executed first, followed by the **Action** value, followed by the **Canvas** value. An executable value associated with one of these three keys in an interest is often referred to as an *executable match.*

The post-match behavior for executable dictionary values is demonstrated by the following example:

```
createevent dup begin                          % Create an event.
   /Name 3 dict dup begin                      % Create dict for the Name field.
      /LeftMouseButton {                       % event => –
         /Action get /UpTransition eq {
               (Left Button Up) ==
         } {
               (Left Button Down) ==
         } ifelse
      } def
      /MiddleMouseButton {                     % event => –
         /Action get /UpTransition eq {
               (Middle Button Up) ==
         } {
               (Middle Button Down) ==
         } ifelse
      } def
      /RightMouseButton {                      % event => –
         /Action get /UpTransition eq {
               (Right Button Up) ==
               exit
         } {
               (Right Button Down) ==
         } ifelse
      } def
   end def
   /Exclusivity true def
end expressinterest                            % Express interest in the event.
                                               % Print instructions.
(Try pressing the left and middle mouse buttons. \n) print
(Then press the right mouse button to exit. \n) print

{
   awaitevent
} loop
```

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

When any mouse button is pressed or released, the server generates an event and distributes a copy of it to the process. After **awaitevent** places the event on the process' operand stack, the executable dictionary key value associated with the event's **Name** is executed immediately, printing the appropriate string to the screen. Run this example with `psh` and press the three mouse buttons to see how it works.

NOTE *Dictionaries with executable values, which are permitted in the* **Canvas, Name,** *and* **Action** *keys of interests, provide a highly efficient way of executing code according to the interest that has been matched. This feature avoids the need for constructs such as* **case**, *which would otherwise be required to direct a matched event to the correct handler. (For a description of the* **case** *utility, see Chapter 11, "Extensibility through NeWS Procedure Files.")*

4.5. Event Distribution: Matching an Event to Multiple Interests

This section describes how the server searches through the currently expressed interests to find matches for the event most recently removed from the head of the global event queue. It also describes how you can restrict this search. A code example of multiple interest matching is given in the last subsection.

Canvas Interest Lists

Each interest that is expressed is either a *pre-child* interest or a *post-child* interest. Each canvas has an interest list which contains its *pre-child* interests followed by its *post-child interests.* When an interest is expressed, it is assigned to the interest list of one or more canvases. The interest list of a canvas can be retrieved with the canvas' **Interests** key (the key's value is an array that holds the canvas' interests). This subsection describes how interests are assigned to canvas interest lists; the next subsection describes how the server searches canvas interest lists to find interests that match the event it is trying to distribute.

NOTE *Each NeWS process also has an interest list. The list contains all interests currently expressed by the process. The interest list of a process can be retrieved with the process'* **Interests** *key. The process interest lists are not used during the server's search for matching interests; they merely provide a convenient way to see the current interests of any particular process.*

Pre-Child and Post-Child Interests

The event dictionary contains a key named **IsPreChild**. This key's value is only meaningful for interests. When the key's value is set to **true** in an interest, the interest is a *pre-child* interest. When the value is set to **false**, the interest is a *post-child* interest. The value of the **IsPreChild** key defaults to **false**, making interests post-child by default. If a process wants a pre-child interest, it must set the interest's **IsPreChild** key to **true** before expressing the interest.

Assigning Interests to Canvas Interest Lists

The event dictionary contains a **Canvas** key, whose value can be a canvas, a dictionary or array that contains canvases, or the **null** value. An interest is assigned to one or more canvas interest lists based on the value of its **Canvas** key:

- When an interest is expressed with a single canvas specified as its **Canvas** key value, the interest is inserted into the interest list of the specified canvas.
- When an interest is expressed with an array or dictionary specified as the value of its **Canvas** key, the interest is inserted into the interest list of each canvas in the array or dictionary.
- When an interest is expressed with a **null** value for its **Canvas** key, the interest is inserted into the pre-child interest list of the global root canvas (regardless of the value of its **IsPreChild** key).

Interest List Order

The canvas interest list is used during the server's search for matching interests. The interest list order is important because the server searches a canvas interest list from the head of the list to the tail of the list. When an interest is added to a canvas interest list, it is sorted into the list according to the following rules:

- First, pre-child interests are placed before post-child interests in the canvas' interest list.
- Within the pre-child and post-child parts of the interest list, higher priority interests are placed before lower priority interests. An interest's priority is given by the value of its **Priority** key; the value can be any number, including negative or fractional values. The default priority value is zero. Larger numbers indicate higher priority.
- Among interests with the same priority, exclusive interests are placed before non-exclusive interests. An interest's exclusivity is given by the value of its **Exclusivity** key, which can be **true** or **false**. The default value is **false**. (See the subsection "Exclusive Interests" for a description of the **Exclusivity** key's role in the interest matching process.)
- Among exclusive interests of the same priority, more recently expressed interests are placed before less recently expressed interests.
- Among non-exclusive interests of the same priority, more recently expressed interests are placed before less recently expressed interests.

Thus, the newest, exclusive, pre-child interest with highest priority is always the first interest in that list with which a distributed event is compared and may thus be the first interest in that list that is matched; the oldest, non-exclusive, post-child interest with lowest priority is always the last interest in that list with which the event is compared. The search procedure is described in detail in the next subsection.

Order of Interest Matching: Searching the Canvas Hierarchy

When an event is distributed, the server does not necessarily search all the interests of all the processes. An event is usually only relevant to certain canvases. Therefore, the server only searches the interest lists of the relevant canvases. The value in the event's **Canvas** key determines which canvas interest lists are searched for potential matches. The exact search path through the canvas hierarchy depends on whether the event's **Canvas** value is a single canvas, an array or dictionary containing multiple canvases, or **null**. These search paths are described below.

NOTE *The search for matching interests is subject to restrictions imposed by the canvas* **EventsConsumed** *key and the event* **Exclusivity** *key, described in later subsections; the search procedure described below may be stopped at any time by one of these keys.*

If the Event's **Canvas** Value is a Single Canvas

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

1. The server searches the **pre-child** interests of each canvas on the branch of the canvas hierarchy connecting the global root canvas to the specified canvas. This pre-child interest search starts with the pre-child interests of the

global root canvas and continues through the pre-child interests of the specified canvas.

2. The server searches the post-child interests of the specified canvas.

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

If the Event's **Canvas** Value is an Array or Dictionary

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

If the Event's **Canvas** Value is **null**

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

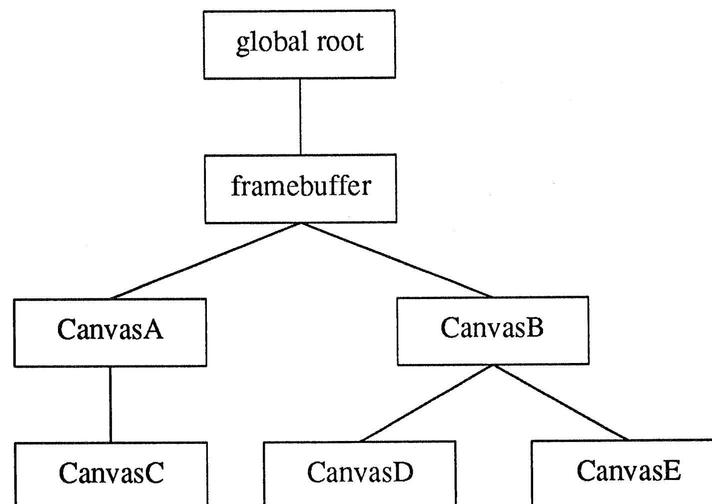
1. The server searches the **pre-child** interest list of each canvas on the branch of the canvas hierarchy that connects the global root canvas to the topmost (leafmost) canvas under the (x, y) location specified in the event. This pre-child interest search starts with the global root canvas and ends with the topmost canvas under the event's location.
2. The server searches the **post-child** interests of each canvas on the branch, starting with the topmost canvas under the event's location and ending with the global root canvas.

Therefore, when **null** is specified as an event's **Canvas** key value, the server searches all pre-child and post-child interests of canvases in the search path.

Search Path Example

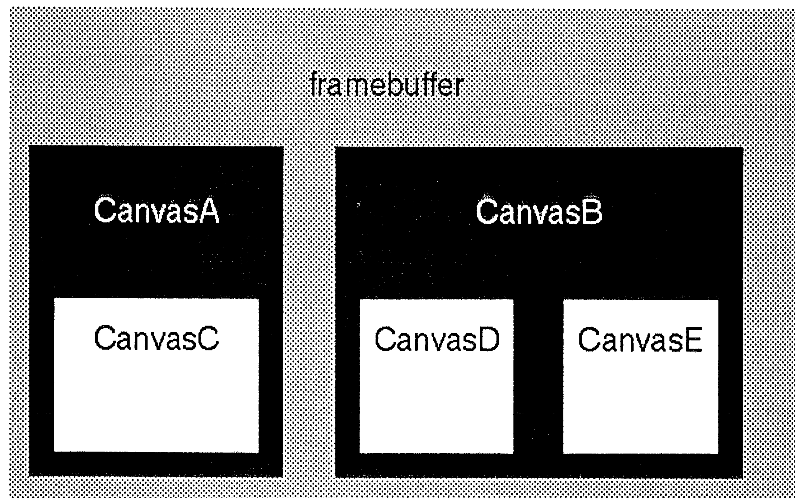
The diagram below illustrates an example canvas hierarchy. The hierarchy includes two children and three grandchildren of the framebuffer canvas.

Figure 4-3 *Example canvas hierarchy*



The following figure illustrates how these example canvases might appear on the screen:

Figure 4-4 *Example canvas hierarchy as it might appear on the screen*



If you press a mouse button over CanvasD, the server sends a mouse button event to the global event queue. This event has **null** in its **Canvas** key. The event's **Coordinates** key contains the (x, y) coordinates of the pointer at the time the mouse button was pressed. When the event is removed from the global event queue for distribution, the server performs the following search for matching interests:

- The pre-child interests of the global root canvas, the framebuffer canvas, CanvasB, and CanvasD are searched (in that order).
- The post-child interests of CanvasD, CanvasB, the framebuffer canvas, and the global root canvas are searched (in that order).

If a process sends an event with CanvasC in its **Canvas** key, the server performs the following search for matching interests:

- The pre-child interests of the global root canvas, the framebuffer canvas, CanvasA, and CanvasC are searched (in that order).
- The post-child interests of CanvasC is searched.

Stopping the Search

The search procedure described in the previous section can be stopped at any time by a canvas' **EventsConsumed** key or an interest's **Exclusivity** key. This section discusses ways in which the search for matching interests can be stopped.

Canvas Event Consumption

The **canvastype** dictionary contains a key named **EventsConsumed** that affects the testing of an event against post-child interests; the key specifies whether events tested for a match with the canvas' post-child interests are tested with the post-child interests of the canvas' parent. The following list describes the three possible values for a canvas' **EventsConsumed** key:

- **/AllEvents**

  This value indicates that all events tested for a match with the canvas' post-child interests are consumed; they are not tested for a match with the post-child interests of the canvas' ancestors.

- **/MatchedEvents**

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

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

- **/NoEvents**

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

Non-consumed events are tested against the post-child interests of the canvas' grandparent depending on the **EventsConsumed** status of the canvas' parent. Thus, if all canvases in a branch extending to the global root canvas have their **EventConsumed** keys set to **/NoEvents**, all events are tested against all post-child interests of each canvas; this assumption was made for the example in the previous subsection "Search Path Example."

Exclusive Interests

The **eventtype** dictionary contains an **Exclusivity** key. This key, which holds a boolean value, is significant only for interests; its value is ignored in distributed events. If the value of an interest's **Exclusivity** key is **true**, a distributed event successfully matched with the interest is not compared with any more interests. Thus, the **Exclusivity** key can be used to consume events during pre-child or post-child testing.

Redistributing an Event Stopped by an Exclusive Interest

The following operator allows you to override the exclusivity of an interest:

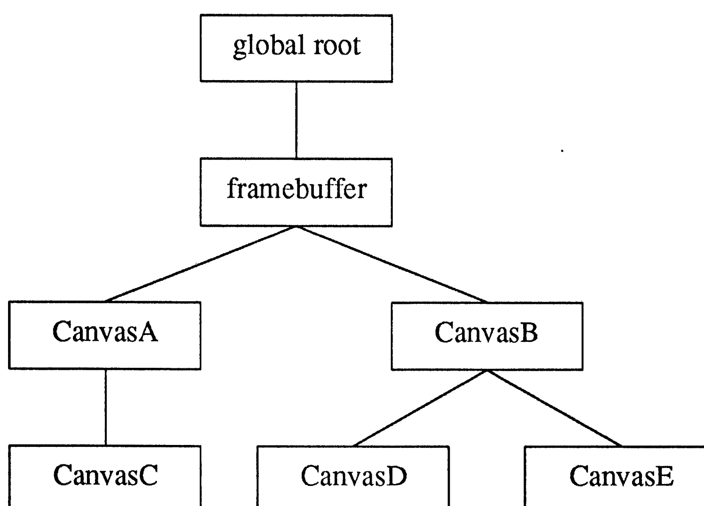
event **redistributeevent** –

This operator resumes the distribution of *event*, where *event* is an event that has matched an exclusive interest and has been returned by **awaitevent**. The **redistributeevent** operator continues the distribution process; **redistributeevent** does not reinsert *event* into the global event queue. No interest compared with *event* since the last call to **sendevent** is allowed to match *event* again (see Section 4.3.3, "Rules for Matching **Serial** Key Values").

Modified Search Path Example

This discussion uses the same canvas hierarchy as the previous "Search Path Example" subsection, but this discussion shows how the **EventsConsumed** and **Exclusivity** keys affect the search. The canvas hierarchy is shown again in the following figure:

Figure 4-5 *Example canvas hierarchy*



If a mouse button is pressed over CanvasD, the following scenarios are possible:

- The event could be compared with all pre-child and post-child interests of the global root canvas, the framebuffer canvas, CanvasB, and CanvasD. This scenario is possible if the framebuffer canvas, CanvasB, and CanvasD have their **EventsConsumed** fields set to **/NoEvents**, and if none of the matching interests have their **Exclusivity** keys set to **true**.

- The search could be stopped at any time by an exclusive interest, even before the search gets to CanvasD. For example, if CanvasB has a matching, exclusive, pre-child interest, the search would stop after checking the pre-child interests for the global root canvas, the framebuffer canvas, and CanvasB.

- The search could be stopped at some point during testing against post-child interests if the event matches an interest whose canvas has its **EventsConsumed** key set to **/MatchedEvents**.

  For example, the event could be compared with all the pre-child interests on the search path, but only with the post-child interests of CanvasD. This scenario would occur if CanvasD has a matching post-child interest and CanvasD's **EventsConsumed** key is set to **/MatchedEvents**. This scenario is common because the default value for a canvas' **EventsConsumed** key is **/MatchedEvents**.

- The event could be stopped at some point during testing against post-child interests if CanvasD or CanvasB has an **EventsConsumed** value of **/AllEvents**; in this case, the interest does not even have to match the event for the search to stop. For example, if CanvasD consumes all events, an

event that is tested against CanvasD's post-child interests will not be tested against the post-child interests of CanvasB, the framebuffer canvas, or the global root canvas.

Hints for Using Pre-Child and Post-Child Interests

You should use post-child interests instead of pre-child interests whenever possible because pre-child interests have higher performance costs. Even if an event is sent to a specific canvas, all the pre-child interests along the search path are checked for potential matches. Minimizing the number of pre-child interests reduces this search time.

You will need to use pre-child interests in some cases. For example, suppose you have a child canvas with a parent frame canvas beneath it. Also suppose that the child canvas is used for text entry and that you have a click-to-type keyboard focus convention. You might want the frame canvas to highlight itself when the user clicks on the child canvas; the frame's change of color would indicate that the child canvas had become the keyboard focus. In this case, you would want the frame canvas to have a pre-child interest for mouse presses. The child canvas might consume the mouse press event, or its interest in mouse presses might be exclusive; therefore, the frame canvas needs a pre-child interest in button presses to assure that it will receive button press events.

NOTE

Events are often directed at a particular canvas (for example, damage events) or are only relevant to certain canvases (for example, input events). Therefore, a canvas is often said to "receive" an event, or an event is said to be "sent" to a canvas. These phrases are used for convenience. Strictly speaking, the event is sent to and received by the process that holds the matching interest; the canvas just has the matching interest on its canvas interest list, allowing the event to be distributed to the process. After the process receives an event, it usually takes an action based on the needs of the canvas whose interest was matched.

Example: Matching Multiple Interests

The following example demonstrates how an event is matched with more than one interest:

```
/MakeCanvas {  % color x y w h parent => canvas
    newcanvas
    5 1 roll newpath rectpath
    dup reshapecanvas dup setcanvas
    dup begin
        /Mapped true def
        /EventsConsumed /NoEvents def
        /Transparent false def
    end
    exch fillcanvas
} def

/paintfb {                                  % Damage entire
    framebuffer setcanvas                   % framebuffer so
    clippath extenddamage                   % server will repaint it.
    pause
} def
```

```
/CanvasA 0 20 20 200 260 framebuffer MakeCanvas def

/CanvasB 0 260 20 320 260 framebuffer MakeCanvas def

/CanvasC 1 40 40 160 120 CanvasA MakeCanvas def

/CanvasD 1 280 40 120 120 CanvasB MakeCanvas def

/CanvasE 1 440 40 120 120 CanvasB MakeCanvas def

/rendertext { % event canvas x y => -
    3 index /TimeStamp get time ne {           % Repaint if needed.
        CanvasA setcanvas 0 fillcanvas         % (Only repaint if
        CanvasB setcanvas 0 fillcanvas         % event is first
        CanvasC setcanvas 1 fillcanvas         % one for that
        CanvasD setcanvas 1 fillcanvas         % button press.)
        CanvasE setcanvas 1 fillcanvas
        paintfb
        3 index /TimeStamp get /time exch def  % Update time.
    } if
    2 index setcanvas                          % Set current canvas.
    moveto                                     % Set current point.
    0 setgray
    dup CanvasA eq { 1 setgray } if
    CanvasB eq { 1 setgray } if
    /Name get cshow                            % Print Name (Got it).
} def

createevent dup begin                          % Express a pre-child
    /Name 1 dict dup begin                     % interest in left
        /LeftMouseButton (Got it) def          % button presses for all
    end def                                    % the canvases.
    /Action /DownTransition def
    /Canvas 6 dict dup begin
        CanvasA      { CanvasA 120 220 rendertext } def
        CanvasB      { CanvasB 420 220 rendertext } def
        CanvasC      { CanvasC 120 120 rendertext } def
        CanvasD      { CanvasD 340 120 rendertext } def
        CanvasE      { CanvasE 500 120 rendertext } def
        framebuffer  { framebuffer 300 350 rendertext } def
    end def
    /IsPreChild true def
end expressinterest

createevent dup begin                          % Express a post-child
    /Name 1 dict dup begin                     % interest in middle
        /MiddleMouseButton (Got it) def        % button presses for all
    end def                                    % the canvases.
    /Action /DownTransition def
    /Canvas 6 dict dup begin
        CanvasA      { CanvasA 120 220 rendertext } def
        CanvasB      { CanvasB 420 220 rendertext } def
        CanvasC      { CanvasC 120 120 rendertext } def
```

```
            CanvasD         { CanvasD 340 120 rendertext } def
            CanvasE         { CanvasE 500 120 rendertext } def
            framebuffer     { framebuffer 300 350 rendertext } def
        end def
    end expressinterest

    createevent dup begin                               % Express interest in
        /Name 1 dict dup begin                          % right button presses.
            /RightMouseButton { pop exit } def          % Exit if right button
        end def                                         % event is returned.
        /Action /DownTransition def
        /Exclusivity true def
    end expressinterest

    /time 0 def
    /Helvetica findfont 26 scalefont setfont
                                                        % Print instructions.

    (Click left and middle mouse buttons over various canvases. \n) print
    (Watch the events that are distributed.  Left presses match \n) print
    (pre-child interests; middle presses match post-child interests. \n) print
    (Press right mouse button to exit loop. \n) print

    { awaitevent .014 sleep } loop                      % Loop, pausing
                                                        % between each event.
    paintfb                                             % Repaint framebuffer.
```

This example uses the same canvas hierarchy as the previous "Search Path Example" and "Modified Search Path Example" subsections. This example sets each canvas' **EventsConsumed** field to **/NoEvents**. After mapping the canvases, this example expresses interest in left, middle, and right button presses. The interest in middle and right button presses are post-child interests; the interest in left button presses is pre-child. When you press the left or middle mouse button over any canvas, the string "Got it" is printed in every canvas that receives that event. If you press the right mouse button, you will exit the **awaitevent** loop and quit the example.

Notice that this example uses a dictionary with executable values for the **Canvas** field of the left and middle button press interests. The executable values set the current point to a location appropriate for the canvas whose interest was matched. Then the string "Got it" is printed at the current point.

Try this example; press the left and middle buttons over various canvases and observe the distribution of each event. You can see the distribution of a left press event make its way down the pre-child interests in the canvas hierarchy, and the distribution of a middle press event make its way up the post-child interests.

If you are running the default window manager, `olwm`, a middle mouse button press does not cause the string "Got it" to be printed on the framebuffer canvas. As an X11 client, the `olwm` window manager expresses the equivalent of an exclusive post-child interest in button events on the framebuffer canvas; therefore, clients will not receive those events. If you are running some other window

manager, you may see "Got it" printed on the framebuffer canvas when you press the middle mouse button. See the *X11/NeWS Server Guide* for more information about window managers.

The following three figures illustrate the pre-child interests matched when the left mouse button is pressed over CanvasD. (For clarity, the canvases are labeled A through E in these figures. However, for simplicity, the labeling code is not included in the example code given above; you will not see the labels on your screen when you run this example.)

Figure 4-6 *The first pre-child interest matched*

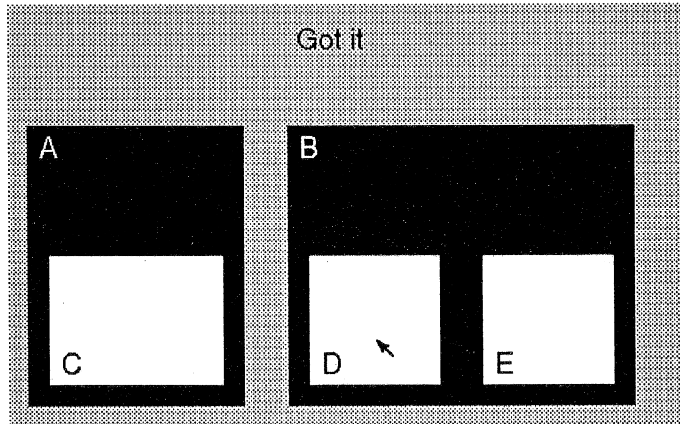


Figure 4-7 *The second pre-child interest matched*

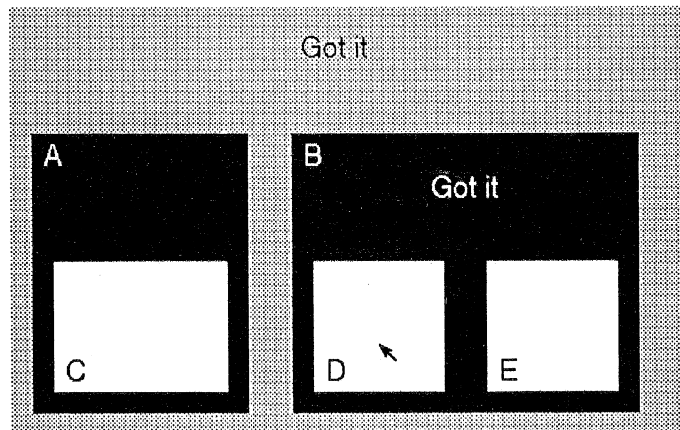
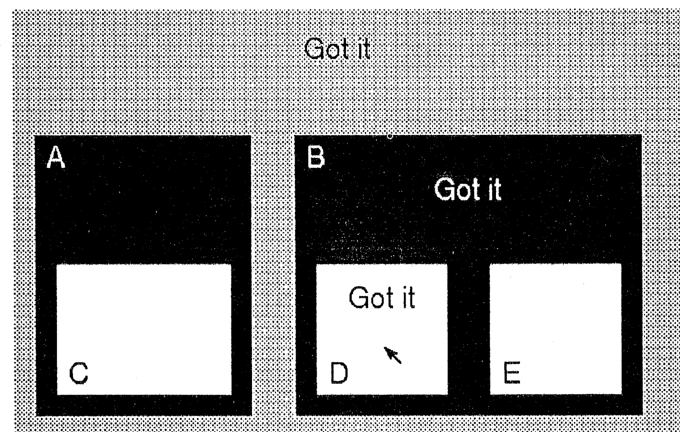


Figure 4-8 *The third pre-child interest matched*



The next two figures illustrate the post-child interests matched when the middle mouse button is pressed over CanvasC:

Figure 4-9 *The first post-child interest matched*

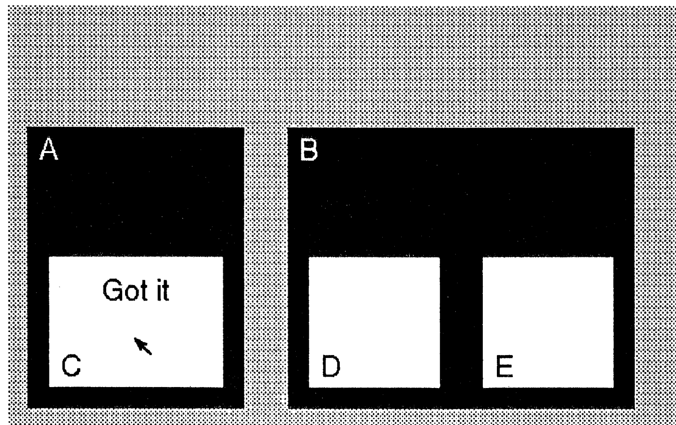
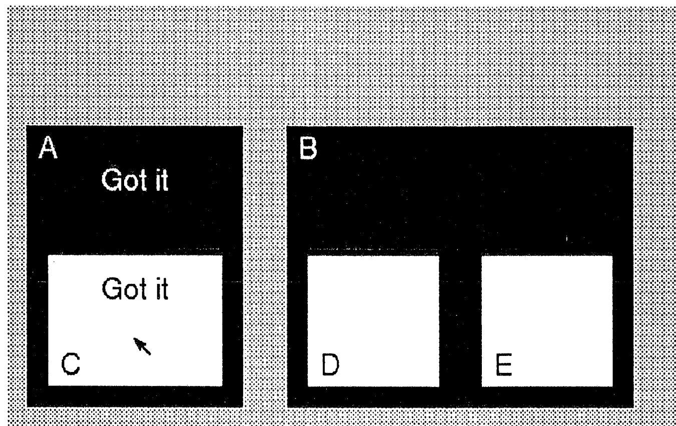


Figure 4-10 *The second post-child interest matched*



Now, edit your file to make the **EventsConsumed** field of CanvasC be **/MatchedEvents**. You can make this change by adding the following line to the code after CanvasC is defined:

```
CanvasC /EventsConsumed /MatchedEvents put
```

Now when you run the example and press the middle mouse button over CanvasC, only the post-child interest of CanvasC is matched; the distribution of the middle button press event is halted after the event matches the post-child interest of CanvasC. Left button presses are still distributed to CanvasC's ancestors because they match pre-child interests; the **EventsConsumed** field only affects testing against post-child interests.

Now, edit the code file again. Add the following line after CanvasD is defined:

```
CanvasD /EventsConsumed /AllEvents put
```

Also, comment out the line that assigns CanvasD a key/value pair in the **Canvas** field of the middle button press interest. The interest should now appear as follows:

```
createevent dup begin                                % Express a post-child
    /Name 1 dict dup begin                           % interest in middle
        /MiddleMouseButton (Got it) def              % button presses for all
    end def                                          % the canvases.
    /Action /DownTransition def
    /Canvas 6 dict dup begin
        CanvasA       { CanvasA 120 220 rendertext } def
        CanvasB       { CanvasB 420 220 rendertext } def
        CanvasC       { CanvasC 120 120 rendertext } def
        %CanvasD      { CanvasD 340 120 rendertext } def
        CanvasE       { CanvasE 500 120 rendertext } def
        framebuffer   { framebuffer 300 350 rendertext } def
    end def
end expressinterest
```

Now when you run the example and press the middle mouse button over CanvasD, no events are distributed. CanvasD no longer has an interest in middle button presses, and its event consumption of **/AllEvents** prevents the middle button press from matching the post-child interests of CanvasD's ancestors. If you press the middle button over CanvasB, you will see that the post-child interest of CanvasB is still matched.

Now edit the file again. Give CanvasB a high priority pre-child interest in left button presses by adding the following lines just before the **awaitevent** loop:

```
createevent dup begin
    /Name 1 dict dup begin
        /LeftMouseButton (Got it) def
    end def
    /Action /DownTransition def
    /Canvas 1 dict dup begin
        CanvasB { CanvasB 420 190 rendertext } def
    end def
    /Priority 1 def
    /IsPreChild true def
end expressinterest
```

When you run the example and press the left mouse button over CanvasB, CanvasD, or CanvasE, the higher priority interest for CanvasB is matched before the lower priority interest for CanvasB. Thus, the string Got it is printed twice in CanvasD, the first time lower on the screen than the second time.

Now edit the file one last time. Replace the line

/Priority 1 def

with the line

```
/Exclusivity true def
```

Now when you run the example and press the left mouse button over CanvasD or CanvasE, only the pre-child interest of the framebuffer canvas and the exclusive pre-child interest of CanvasB are matched; the exclusive interest of CanvasB prevents left button press events from being distributed to CanvasB's descendants.

4.6. System-Generated Events

The server automatically creates and sends a *system-generated* event in the following circumstances:

- A keyboard key is pressed.
- An object becomes obsolete and its memory needs reclaiming.
- A process dies while it is still referenced or while **waitprocess** is being executed on it.
- The mouse is dragged or a mouse button is pressed.
- The mouse pointer exits one canvas and enters another.
- The keyboard focus exits one canvas and enters another.
- A canvas is damaged for the first time since its last repair.

System-generated events are sent into distribution by the server, but once these events enter the global event queue, they are treated no differently than process-generated events; the NeWS operators for expressing interest and awaiting events must be used for system-generated events in the same way as is required for process-generated events.

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

Keyboard Events

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

Obsolescence Events

Obsolescence events are generated by the server for an object that becomes obsolete. *Obsolescence* is defined as the state in which all the references to an object are *soft*. (See the discussion of *soft references* in Chapter 8, "Memory Management".) The value of the event's **Name** field is **/Obsolete** and the value of the event's **Action** field is the obsolete object.

ProcessDied Events

A *ProcessDied* event is generated if a lightweight process dies when references to it exist or a **waitprocess** is being executed upon it. The value of the event's **Name** key is **/ProcessDied** and the value of the **Action** key is the process itself.

The following interactive `psh` example demonstrates a **ProcessDied** event:

```
% psh
executive
Welcome to X11/NeWS Version 2.1

createevent dup                          % Express interest in
/Name /ProcessDied put                   % ProcessDied events.
expressinterest
                                         % Fork process that names itself.
{ currentprocess /ProcessName (ChildProcess) put } fork

pstack                                   % Print operand stack.
process (0x210efc, 'ChildProcess', zombie, null)

awaitevent                               % Retrieve event from local queue.
dup /Name get ==                         % Get event's Name.
/ProcessDied
/Action get /ProcessName get ==          % Get the ProcessName.
(ChildProcess)
pop                                      % Pop child process.
quit                                     % Quit psh.
```

Mouse Events

The server automatically generates *mouse events* when the user manipulates the mouse. The server assigns appropriate values to the event's **Name, Action, Coordinates, XLocation**, and **YLocation** keys; the value of the **Canvas** key is always set to **null**. Mouse events are generated in the following circumstances:

- The mouse is dragged.

  The value of the event's **Name** key is set to **/MouseDragged**, and the value of the **Action** key is set to **null**. The values of the **Coordinates, XLocation**, and **YLocation** keys are set to the new location of the mouse pointer.

  The server keeps generating **/MouseDragged** events as long as the user keeps moving the mouse. Thus, a certain number of discrete events are generated to report a user action that is continuous. The number of events generated for any particular mouse drag is system dependent.

- A mouse button is pressed and released.

  When the mouse button is pressed, the value of the event's **Name** key is set to **/LeftMouseButton, /MiddleMouseButton**, or **/RightMouseButton**, depending on which button is pressed; the value of the **Action** key is set to **/DownTransition**. When the button is released, another event is generated with the same **Name** value and with the **Action** set to **/UpTransition**. Thus, two events are automatically generated whenever a mouse button is pressed and released. For each event, the values of the **Coordinates, XLocation**, and **YLocation** keys are set to the location of the mouse pointer at the time of the mouse press or release.

The following example demonstrates mouse events:

```
%
% Create canvas to play in.
%
/MyCanvas framebuffer newcanvas def     % Create a canvas object.
25 25 translate                         % Move its origin.
0 0 400 400 rectpath                    % Make a rectangular path.
MyCanvas reshapecanvas                  % Make our canvas that shape.
MyCanvas /Mapped true put               % Map the canvas.
MyCanvas setcanvas                      % Make canvas the currentcanvas.
1 fillcanvas                            % Give it a white background.
0 setgray                               % Draw with black lines.
3 setlinewidth

%
% Print (in the canvas) documentation
% on button usage.
%
/Times-Roman findfont 16 scalefont setfont
10 40 moveto
(Press left button to move currentpoint) show
10 25 moveto
(Press middle button and drag to draw a line) show
10 10 moveto
(Press right button to quit) show
200 200 moveto                          % set starting point.

%
% Create an interest in MouseDragged events on our play canvas
% (store in /drag); this is an executable match that draws a
% line to the current mouse position each time the mouse moves
% while this interest is expressed. It also leaves the
% currentpoint at the mouse position.
%
/drag createevent dup begin
   /Name 1 dict dup begin
      /MouseDragged {                   % event => -
         begin
            XLocation YLocation lineto stroke  % Consumes the path.
            XLocation YLocation moveto          % Set currentpoint to same.
         end
      } def
   end def
   /Action null def
   /Canvas MyCanvas def
end def

%
% Create an interest in Up and Down transitions of all
% three mouse buttons. Each button has its own handler
% associated with it (the value of the corresponding key
% in the /Name field of the interest).
%
createevent dup begin
```

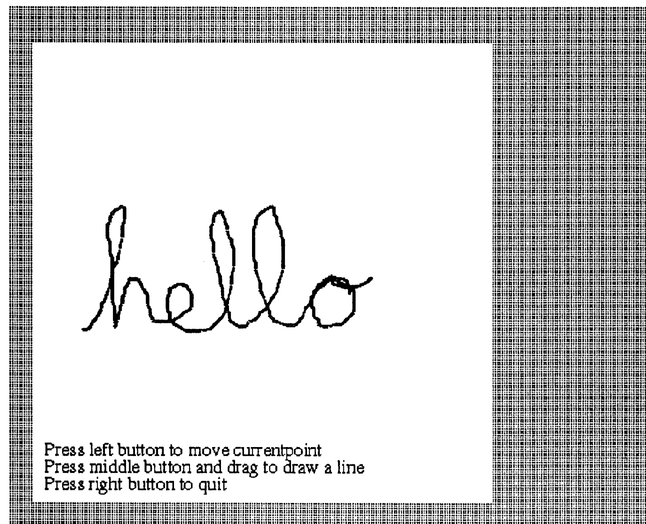
```
    /Name 3 dict dup begin
      /LeftMouseButton {                        % event => –
        begin
          XLocation YLocation moveto            % Move the currentpoint.
        end
      } def
      /MiddleMouseButton {                      % event => –
        begin
          Action /DownTransition eq {
            drag expressinterest                % We want drag events now.
            XLocation YLocation lineto stroke   % Stroke consumes the path.
            XLocation YLocation moveto          % So set currentpoint back.
          } {
            drag revokeinterest                 % Don't want drag events any more.
          } ifelse
        end
      } def
      /RightMouseButton {                       % event => –
        /Action get                             % We're all done...
        /UpTransition eq {                      % Break out of the {} loop.
          exit
        } if
      } def
    end def
    /Action [ /DownTransition /UpTransition ] def
    /Canvas MyCanvas def
  end expressinterest

  { awaitevent } loop                           % Loop, processing events.
```

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

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

Try running this example with `psh` and drawing in the canvas that is generated. The figure below shows an example in which the word "hello" was drawn on the canvas by dragging the mouse while pressing the middle mouse button.

Figure 4-11 *An example of drawing in the canvas*



Enter and Exit Events

The server generates canvas crossing events whenever the mouse pointer moves from one canvas to another. Each such event is directed to a particular canvas, identified in the event's **Canvas** field; the event specifies how the pointer moved with respect to that canvas.

The server sets the **Name** key to **/ExitEvent** or **/EnterEvent**, depending on the movement of the pointer with respect to the canvas. The server sets the **Action** key to the numeric value 0, 1, 2, 3, or 4, depending on the hierarchical relationships between the canvas that receives the event, the canvas from which the pointer moves (the "source"), and the canvas into which the pointer moves (the "destination").

When the pointer crosses any canvas boundary, at least two events are generated: an exit event for the canvas being exited and an enter event for the canvas being entered. More than two events may be generated if the source and destination canvases do not have a parent/child relationship. The pointer motion scenarios can be categorized in three main groups:

- The pointer moves from a child canvas to its parent or from a parent canvas to its child. The source canvas receives an exit event, and the destination canvas receives an enter event.
- The pointer moves between two canvases, one of which is a descendant, but not a child, of the other. The source canvas receives an exit event, the destination canvas receives an enter event, and all canvases between the source and destination also receive events; the intermediate canvases receive exit events if the source is a descendant of the destination, and they receive enter events if the source is an ancestor of the destination.
- The pointer moves between two canvases that are not on the same branch of the canvas hierarchy. The source canvas receives an exit event, the destination canvas receives an enter event, and all canvases on both branches up to but not including the *least common ancestor* of the source and destination

also receive events. The least common ancestor is the canvas at the junction of the two branches. Ancestors of the source canvas receive exit events, and ancestors of the destination canvas receive enter events.

The following table describes the **Action** values for enter and exit events. Note that a canvas is said to contain the pointer *directly* when it is the topmost canvas under the pointer; a canvas is said to contain the pointer *indirectly* if it is an ancestor of a canvas that *directly* contains the pointer. Note also that a canvas does not receive a crossing event if it contains the pointer directly both before and after the pointer movement, nor does it receive a crossing event if it contains the pointer indirectly both before and after the pointer movement. When the phrase "the canvas" or "this canvas" is used in the following table, it refers to the canvas that receives the crossing event.

Table 4-1 *Action Values for Enter and Exit Events*

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

Table 4-1 *Action Values for Enter and Exit Events— Continued*

Name	*Action*	*Explanation*
	3	The canvas formerly contained the pointer *directly*; the new direct container is not an ancestor or descendant of this canvas.
	4	The canvas formerly contained the pointer *indirectly*; the new direct container is not an ancestor or descendant of this canvas.

The following example demonstrates enter and exit events. The example uses the same canvas hierarchy as was used in the examples for multiple interest matching, but the canvases are given different shapes and positions. After mapping the canvases, the example expresses interest in enter and exit events for all the canvases.

When you run this example and move the mouse from one canvas to another, the type of event that each canvas receives is written in that canvas; the string "Enter" or "Exit" is written, and the numeric **Action** value is written beneath the string. When you move the mouse to generate the next set of crossing events, all the canvases are automatically repainted before the names and actions are printed. Run this example with `psh`; move the mouse around the canvas hierarchy and observe the types of enter and exit events that are generated. Press the right mouse button to quit.

The example code is given below.

```
/MakeCanvas {  % color x y w h parent => canvas
    newcanvas
    5 1 roll newpath rectpath
    dup reshapecanvas dup setcanvas
    dup begin
        /Mapped true def
        /EventsConsumed /NoEvents def
        /Transparent false def
    end
    exch fillcanvas
} def

/paintfb {                                  % Damage entire
    framebuffer setcanvas                   % framebuffer so
    clippath extenddamage                   % server will repaint it.
    pause
} def

/CanvasA  0  20  40  240  260  framebuffer MakeCanvas def
newpath  22  42  236  256  rectpath clipcanvas 0.75 fillcanvas

/CanvasB  0  240  20  340  260  framebuffer MakeCanvas def
newpath  242  22  336  256  rectpath clipcanvas 1 fillcanvas
```

```
/CanvasC 0 80 40 180 140 CanvasA MakeCanvas def
newpath 82 42 176 136 rectpath clipcanvas 0.9 fillcanvas

/CanvasD 0 240 20 180 120 CanvasB MakeCanvas def
newpath 242 22 176 116 rectpath clipcanvas 0.75 fillcanvas

/CanvasE 0 360 40 200 120 CanvasB MakeCanvas def
newpath 362 42 196 116 rectpath clipcanvas 0.88 fillcanvas

/rendertext { % event x y => -
    2 index /TimeStamp get time ne {          % Repaint if needed.
        CanvasA setcanvas 0.75 fillcanvas
        CanvasB setcanvas 1 fillcanvas
        CanvasC setcanvas 0.9 fillcanvas
        CanvasD setcanvas 0.75 fillcanvas
        CanvasE setcanvas 0.88 fillcanvas
        paintfb
        2 index /TimeStamp get /time exch def  % Update time.
    } if
    2 index /Canvas get setcanvas              % Set current canvas.
    moveto                                     % Set current point.
    0 setgray
    dup /Name get gsave cshow grestore         % Print Name.
    /Action get 10 string cvs 0 -40 rmoveto cshow  % Print Action.
} def

createevent dup begin                          % Express interest in
    /Name 2 dict dup begin                     % enter and exit events
        /EnterEvent (Enter) def                % for all the canvases.
        /ExitEvent (Exit) def
    end def
    /Canvas 6 dict dup begin
        CanvasA      { 120 240 rendertext } def
        CanvasB      { 420 240 rendertext } def
        CanvasC      { 160 100 rendertext } def
        CanvasD      { 300 100 rendertext } def
        CanvasE      { 460 120 rendertext } def
        framebuffer  { 300 370 rendertext } def
    end def
end expressinterest

createevent dup begin                          % Express interest in
    /Name 1 dict dup begin                     % right button presses.
        /RightMouseButton { pop exit } def     % Exit if right button
    end def                                    % event is returned.
    /Action /DownTransition def
    /Exclusivity true def
end expressinterest

/time 0 def
/Helvetica findfont 26 scalefont setfont

(Move mouse pointer from one canvas to another.  Notice the \n) print
```

```
(events that are generated.  Press right mouse button to exit loop. \n) print

{ awaitevent } loop

paintfb                                          % Repaint framebuffer.
```

The following two figures illustrate the enter and exit events that are generated when you move the mouse pointer from CanvasC to CanvasD:

Figure 4-12 *Mouse cursor over CanvasC*

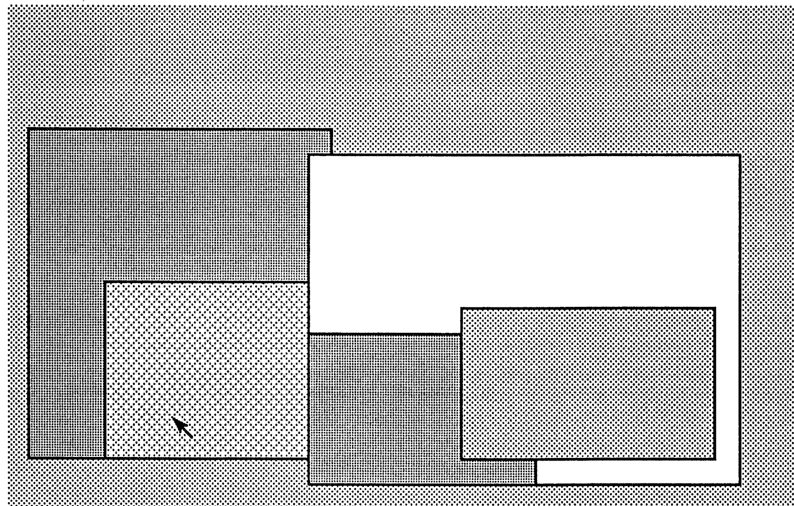
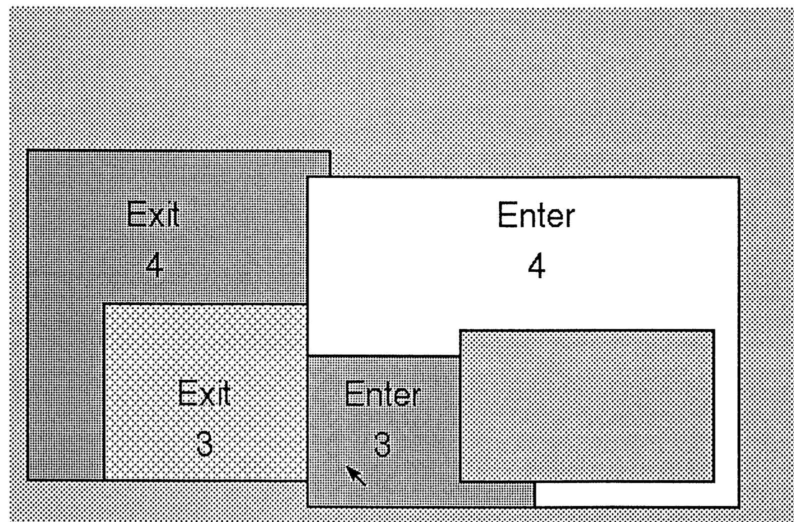
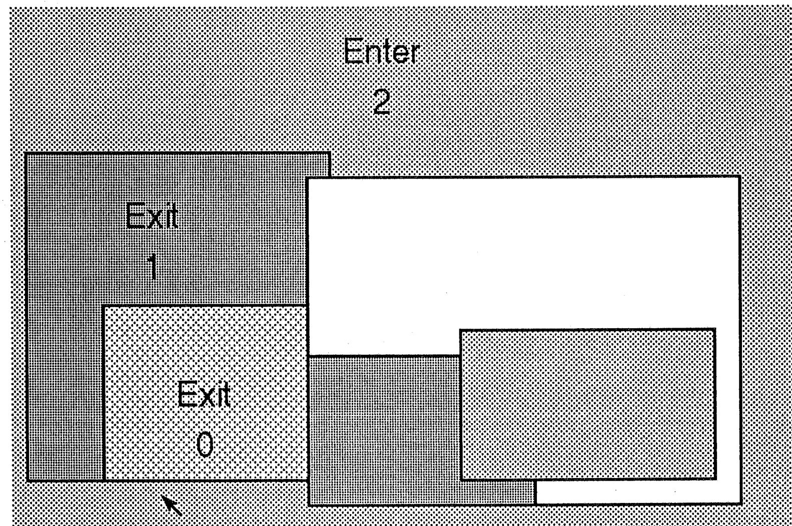


Figure 4-13 *Moving the mouse from CanvasC to CanvasD*



The next figure illustrates the enter and exit events that are generated when you move the mouse pointer from CanvasC to the framebuffer canvas:

Figure 4-14 *Moving the mouse from CanvasC to the framebuffer canvas*



Focus Events

The *keyboard focus* is the canvas that is to receive keyboard input. The user changes the keyboard focus with the mouse. In *click-to-type* mode, the user clicks a mouse button in the canvas that is to become the keyboard focus; in *focus-follows-mouse* mode, the user simply moves the mouse pointer into the canvas that is to become the keyboard focus. The user can choose between these two modes with the **Properties** submenu off of the root menu. The *insertion point* is the location in the keyboard focus canvas at which text will appear when keyboard keys are pressed.

When the user changes the keyboard focus, the server's *focus manager* sends *focus events* to interested processes; a process can express interest in focus events by registering its canvases as clients of the focus manager. When a process receives a focus event indicating that one of its canvases has become the keyboard focus, the process should express interest in keyboard events for that canvas. Keyboard events are not automatically sent to the keyboard focus. Thus, focus events are advisory in nature.

Focus events are canvas crossing events that are similar to mouse pointer events (**/EnterEvent** and **/ExitEvent**), except that focus events indicate that the keyboard focus, rather than the pointer, has shifted from one canvas to another. A focus event is sent to each canvas that loses or gains the keyboard focus (directly or indirectly). A focus event contains the affected canvas in its **Canvas** field. The **Name** of a focus event is always one of the following three values:

- **/RestoreFocus**

  This value indicates that the insertion point has been restored to the position it was in when this canvas was last the focus. This value is used for focus-follows-mouse mode.

- **/AcceptFocus**

  This value indicates that the insertion point has been placed wherever the mouse button was clicked in this canvas. This value is used for click-to-type mode.

- **/LoseFocus**

  This value indicates that the focus has left this canvas.

The **Action** value of a focus event is an integer that specifies the nature of the focus change. The possible values for the **Action** key are described in the following table (this table uses the same terms and conventions as the table given previously for **/Enter** and **/Exit** events):

Table 4-2 *Action Values for Keyboard Focus Events*

Name	*Action*	*Explanation*
/RestoreFocus **/AcceptFocus**	0	The canvas is now the focus; the previous focus was an ancestor of this canvas.
	1	The canvas is now the ancestor of the focus; the previous focus was an ancestor of this canvas.
	2	The canvas is now the focus; the previous focus was a descendant of this focus.
	3	The canvas is now the focus; the previous focus was not an ancestor or descendant of this canvas.
	4	The canvas is now an ancestor of the focus; the previous focus was not an ancestor or descendant of this canvas.
	5	The canvas directly or indirectly contains the pointer and is now a descendant of the focus. The previous canvas is not equivalent to this canvas nor is the previous canvas an ancestor or descendant of this canvas.
	6	The focus is now **/ReDistribute** (this value means that the focus can be any canvas that is currently under the mouse).
	7	The focus is now **None**.
/LoseFocus	0	The canvas was previously the focus; the new focus is an ancestor of this canvas.

Table 4-2 *Action Values for Keyboard Focus Events— Continued*

Name	*Action*	*Explanation*
	1	The canvas was previously an ancestor of the focus; the new focus is an ancestor of this canvas.
	2	The canvas was previously the focus; the new focus is a descendant of this canvas.
	3	The canvas was previously the focus; the new focus is not an ancestor or descendant of this canvas.
	4	The canvas was previously an ancestor of the focus; the new focus is not an ancestor or descendant of this canvas.
	5	The canvas directly or indirectly contains the pointer and was previously a descendant of the focus. The new canvas is not equivalent to this canvas nor is the new canvas an ancestor or descendant of this canvas.
	6	The previous focus was **/ReDistribute** (this value means that the focus can be any canvas that is currently under the mouse).
	7	The previous focus was **None**.

Damage Events

Damage events are generated for a canvas whenever it is *damaged* for the first time since its last repair (a definition of *damage* is provided in Chapter 2, "Canvases"). If a damaged canvas is not repaired immediately and damage continues to occur, the server does not send additional damage events for that canvas. Instead, the damage accumulates. The client can use the **damagepath** operator to retrieve the path that outlines the boundary of the damaged region. The client can then clip to the damage path and repair the damage. When the **damagepath** operator is executed, the damage path is cleared; the next time the canvas is damaged, a damage event is sent and the damage path begins to accumulate again. The value of a damage event's **Action** key is **null**; the value of its **Canvas** key is the canvas that is damaged.

The following example demonstrates damage events. This example maps a parent canvas and its child. You can move the child by clicking the left mouse button. Because both canvases are unretained, they can be damaged when the child is moved. The code expresses interest in damage for both canvases; any damage that occurs is repaired using executable values in a dictionary in the interest's **Canvas** field.

```
/ParentCanvas framebuffer newcanvas def       % Make parent canvas.
25 25 translate
0 0 300 300 rectpath ParentCanvas reshapecanvas
ParentCanvas /Retained false put
ParentCanvas /Mapped true put
ParentCanvas setcanvas 0 fillcanvas

/ChildCanvas ParentCanvas newcanvas def       % Make child canvas.
50 50 translate
0 0 75 75 rectpath ChildCanvas reshapecanvas
ChildCanvas /Transparent false put
ChildCanvas /Mapped true put
ChildCanvas setcanvas 1 fillcanvas

createevent dup begin                         % Express interest in left
    /Name 2 dict dup begin                    % and right button presses.
        /LeftMouseButton {  % ev => –
            begin
                XLocation YLocation
                ChildCanvas setcanvas
                movecanvas
            end
        } def
        /RightMouseButton {  % ev => –
            pop exit
        } def
    end def
    /Action /DownTransition def
    /Canvas ParentCanvas def
end expressinterest

createevent dup begin                         % Express interest in damage
    /Name /Damaged def                        % on both canvases.
    /Canvas 2 dict dup begin                  % Repair damage.
        ParentCanvas {  % ev => –
            pop
            ParentCanvas setcanvas
            damagepath clipcanvas 0 fillcanvas
        } def
        ChildCanvas {  % ev => –
            pop
            ChildCanvas setcanvas
            damagepath clipcanvas 1 fillcanvas
        } def
    end def
end expressinterest

(Press left mouse button to move child canvas. \n) print
(Press right mouse button to quit. \n) print

{ awaitevent } loop
```

Try running this example with psh. Click the left mouse button to move the child canvas around on its parent. The unretained parent receives damage when its child is moved, but the damage is repaired. Note that the damage repair is accomplished by setting the canvas clipping path to the damage path and then painting the damaged region. Note also that you can move the child canvas partially off its parent (causing it to be clipped) and then back onto its parent again, and the damage to both parent and child is repaired.

4.7. Synchronizing Input with Multiple Processes

After the server distributes copies of an event to all interested processes, it removes the next event from the global event queue and begins the search for matching interests. For synchronization purposes, a process may want to ensure that another event is not removed from the global event queue until the process has performed some action based on the previously distributed event. Or, a process might need to block the global event queue until it has time to express its interests. The server provides several methods for achieving these types of synchronization.

Blocking the Global Event Queue with blockinputqueue

A process can execute the **blockinputqueue** operator to suspend the distribution of events from the global event queue:

num *or* null **blockinputqueue** –

This operator prevents events from being removed from the server's global event queue. When the operator is executed, a release time is calculated for the block; the release time is the sum of the current time and the argument to **blockinputqueue**. The argument can be *num* or *null*; *num* is a number in units of 2^{16} milliseconds and *null* represents a system-defined default timeout. When the operator is executed, no event is removed from the global event queue until one of the following has occurred:

- The amount of time specified by the release time has elapsed.
- The **unblockinputqueue** operator is executed.

When nested calls to **blockinputqueue** are made, no event is removed from the global event queue until one of the following has occurred:

- The amount of time specified by the latest of the release times has elapsed.
- The **unblockinputqueue** operator has been executed once for each call to **blockinputqueue**.

Because an event used as the argument to **sendevent** is inserted in the global event queue, its distribution can be inhibited by **blockinputqueue**. However, an event used as the argument to **redistributeevent** is not inserted in the global event queue; thus, its redistribution cannot be inhibited by **blockinputqueue**.

The **unblockinputqueue** operator is described below.

– unblockinputqueue –

This operator releases the event queue lock previously set by **blockinputqueue**. If more than one event queue lock was set, additional calls to **unblockinputqueue** may be required. When all locks are released, events are once again removed from the global event queue for distribution.

The following example demonstrates one use of **blockinputqueue**. In this example, a child process is forked to listen for Message events; if the child receives such an event, it prints the string Got Event! to the screen. After forking the child, the parent sends a Message event and then sleeps. While the parent sleeps, the child runs. Therefore, the Message event is sent to the global event queue before the child has a chance to express interest in it. Unless the global event queue is blocked before sending the Message event, the child will not receive the event (the server discards the event from the global queue if it finds no matching interests). The solution is to block the global event queue before forking the child, and then have the child unblock the queue when it is ready to receive events.

In the code below, the **blockinputqueue** and **unblockinputqueue** operators are commented out. Type the example into a file and run it with these two lines commented out; no Got Event! message is printed to the screen. Then edit your file to uncomment these two lines, and run the example again; the message Got Event! is then printed.

```
/proc {
    createevent begin
        /Name dictbegin
            /Message { (Got Event!\n) print exit } def
        dictend def
    currentdict end expressinterest
    % unblockinputqueue
    { awaitevent } loop
} def

% null blockinputqueue

{ proc } fork

createevent begin
    /Name /Message def
currentdict end sendevent

0.07 sleep killprocessgroup
```

The type of synchronization demonstrated above might be used to implement a menu. Assume that the menu becomes visible when the user presses and holds down the right mouse button. To make a selection from the menu, the user drags

the mouse to the desired menu entry. The menu code expresses interest in right mouse button presses. When it receives a right press event, the menu code maps the menu canvas. Then the code might fork a process that handles the subsequent mouse drags and menu selection. This process expresses interest in mouse drag events and right button releases. When the right button is released, the menu is unmapped. The process must not miss the right button release; otherwise, the menu would remain on the screen even after the user tried to dismiss it. Therefore, the global event queue should be blocked before forking the process, and the process should unblock the queue when it is ready to receive events.

Blocking the Global Event Queue with the Synchronous Key

The **eventtype** dictionary contains a **Synchronous** key that can be used in an interest to provide event synchronization. The value of the **Synchronous** key is a boolean. If an event matches an interest that has its **Synchronous** key set to **true**, the global event queue is blocked; no event is removed from the global event queue until the process that holds the synchronous interest executes the **unblockinputqueue** operator.

This key is especially useful when your code expresses or revokes interest upon the delivery of other events, or when your code changes some aspect of the global state (such as the canvas hierarchy) after receiving other events.

For example, when the user changes the keyboard focus, interest in keyboard events must be expressed for the new focus canvas and revoked for the old focus canvas. If the user is typing continuously on the keyboard both before and after changing the focus, the keystrokes must be directed to the correct canvases; no keystrokes should be missed, and no canvas should receive keystrokes that belong to another canvas.

The necessary synchronization can be achieved by blocking the global event queue when focus events are distributed. The temporary suspension of event distribution gives the clients time to determine which canvases must express or revoke interest in keyboard events. Once the keyboard interest changes are made, the global event queue can be unblocked. The keyboard events are then distributed properly.

The following code example demonstrates this type of situation. Like the example in the previous subsection, this example forks a process that listens for events. The **blockinputqueue** operator is used to ensure that the forked process has time to express its interest before any events are distributed.

In this example, the forked process is interested in receiving events that tell it when to express interest in another type of event. When the process receives an ExpressOtherInterest event, it expresses interest in Message events. Thus, the ExpressOtherInterest events are similar to focus events, and the Message events are similar to keyboard events. When a Message event is received, the forked process prints the string Got Event! to the screen.

In the code below, the Synchronous true def line is commented out, as is the **unblockinputqueue** operator in the expressotherinterest procedure. Type this code into a file and run it with `psh`; when the two lines are commented out, no message is printed to the screen. Then edit your file to uncomment these two lines, and run the example again; the message Got Event! is then printed.

```
/expressotherinterest {
    createevent begin
        /Name dictbegin
            /Message { pop (Got Event!\n) print exit } def
        dictend def
    currentdict end expressinterest
    % unblockinputqueue
} def

/proc {
    createevent begin
        /Name dictbegin
            /ExpressOtherInterest { pop expressotherinterest } def
        dictend def
        %/Synchronous true def
    currentdict end expressinterest
    unblockinputqueue
    { awaitevent } loop
} def

null blockinputqueue
{ proc } fork pop

createevent begin
    /Name /ExpressOtherInterest def
currentdict end sendevent

createevent begin
    /Name /Message def
currentdict end sendevent

0.07 sleep killprocessgroup
```

When the Synchronous true def line is commented out, the forked process never receives the Message event; the server searches for matching interests for the Message event immediately after it distributes the ExpressOtherInterest event, leaving no time for the forked process to express interest in the Message event before it is removed from the global event queue. In this case, the server finds no matching interests for the Message event, so the event is simply removed from the global event queue and no copies are distributed.

When the **Synchronous** key is used, the global event queue is blocked when the forked process' interest matches the ExpressOtherInterest event. A copy of the ExpressOtherInterest event is placed on the local event queue of the forked process, and the executable match in the **Name** key calls the expressotherinterest procedure. This procedure expresses interest in Message events and then unblocks the global event queue, allowing the Message event to be distributed to the forked process. When the Message event is received, the string Got Event! is printed to the screen.

Synchronizing All Events for a Process

The operators described here are intended for clients that were written for earlier versions of NeWS in which synchronization was guaranteed. Clients written for NeWS version 2.1 or later should use the **Synchronous** key for those interests that need synchronization, instead of forcing synchronization of all events.

If a client requires event synchronization for all events delivered to some process, the process can execute the **setcompatinputdist** operator to set its event synchronization mode to **true**:

boolean **setcompatinputdist** –

This operator sets the state of the current process' event synchronization mode. When an event is delivered to a process whose synchronization mode is **true**, the server gives the process a chance to run before the next event is removed from the global event queue. The default synchronization state for new processes is **false**; child processes inherit their parent's synchronization state.

A process can examine the state of its synchronization mode with the **getcompatinputdist** operator:

– **getcompatinputdist** boolean

This operator returns the boolean value of the current process' event synchronization mode.

4.8. Restricting Distribution of an Event to a Specific Process

You can use the **Process** key of an event to restrict the event's distribution to a single process. If you specify a process in an event's **Process** key and then send the event to the server's global event queue with **sendevent**, the server only allows the event to match interests that belong to the specified process. Note that the value of an event's **Process** key does not affect the search procedure described in Section 4.5, "Event Distribution: Matching an Event to Multiple Interests"; the value of the event's **Process** key is simply compared with the value of each interest's **Process** key as part of the matching criteria (see Section 4.3, "Rules for Matching Events to Interests," for a description of the matching rules).

The example below demonstrates the use of the event **Process** key. This example uses a parent process and a child process. Both processes express interest in Message1 and Message2 events. The interests use executable matches in the **Name** key to print a message stating which process received the event and which event was received. As explained in Section 4.7, "Synchronizing Input with Multiple Processes," the **blockinputqueue** operator is used before forking the child process to ensure that the child has time to express its interest before the parent sends the events; the child unblocks the event queue after expressing its interest. The parent sends one Message1 event with the child process specified in its **Process** key, and it sends one Message2 event with a **null** value in the **Process** key.

Type this example into a file and run it with `psh`. You will see that Message1 is received only by the child process, but Message2 is received by both processes. Each process exits its **awaitevent** loop after it receives the Message2 event.

```
createevent dup begin
    /Name dictbegin
        /Message1 { pop (Parent received Message1\n) print } def
        /Message2 { pop (Parent received Message2\n) print exit } def
    dictend def
end expressinterest

null blockinputqueue

/ChildProcess {
    createevent dup begin
        /Name dictbegin
            /Message1 { pop (Child received Message1\n) print } def
            /Message2 { pop (Child received Message2\n) print exit } def
        dictend def
    end expressinterest
    unblockinputqueue
    { awaitevent } loop
} fork def

createevent dup begin
    /Name /Message1 def
    /Process ChildProcess def
end sendevent

createevent dup begin
    /Name /Message2 def
end sendevent

{ awaitevent } loop
```

When you run this example, the following strings are printed to the screen:

```
% psh filename
Child received Message1
Child received Message2
Parent received Message2
```

Note that the interests expressed in this example are placed on the pre-child interest list of the global root canvas because the interests have **null** in their **Canvas** keys. Because the interests are not exclusive and have default priority, they are ordered in the global root canvas' pre-child interest list according to when they are expressed; more recently expressed interests are placed before less recently expressed interests. The child process' interests are more recently expressed and, therefore, they are placed before the parent process' interests in the pre-child interest list of the global root canvas. Thus, the child's interest in Message2 events is matched before the parent's interest in Message2 events.

Also note that the interests have the default value of zero in their **XLocation** and **YLocation** keys. Therefore, the search path through the canvas hierarchy

includes all the canvases on the branch of the canvas hierarchy that connects the global root canvas to the canvas directly under (0, 0). This example does not create any canvases, but if a canvas is located at (0, 0), its interest lists are searched. (However, only the global root canvas has interests in Message1 and Message2 events; therefore, no other matching interests are found.) You could ensure that only the global root canvas' interest list is searched by making the interests exclusive.

4.9. Creating an Event-Logger Process

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

process *or* null **seteventlogger** –

The specified *process* becomes the event-logger. The *process* argument must be a process that has expressed some interest and has entered an **awaitevent** loop. The expressed interest, which must not match any distributed event, is required to prevent **awaitevent** from returning an error. A copy of each event either removed from the global event queue or redistributed with **redistributeevent** is given to the event-logger process before it is given to any other process. The existence of the event-logger does not affect the normal running of the event distribution mechanism.

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

The file `eventlog.ps`, which is described in Chapter 11, "Extensibility through NeWS Procedure Files," provides a formatted display of events that can be used in the context of the **seteventlogger** operator.

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

– **geteventlogger** process *or* null

This operator returns the process that is the current event-logger or **null** if there is no event-logger.

The following example shows how to set an event-logger; it sets the current process to be an event-logger that simply prints the **Name**, **Canvas**, and **Serial** values of left mouse button events. The **awaitevent** loop is exited if a right button event is returned.

```
createevent dup begin                    % Express an arbitrary interest
    /Name -1 def                         % that won't ever be matched;
end expressinterest                      % prevents syntax error in later
                                         % call to awaitevent.
(Press left mouse button several times. \n) print
(Press right mouse button to quit. \n) print

currentprocess seteventlogger            % Create event-logger.
{
    awaitevent begin
        Name /LeftMouseButton eq {
            ( % % % \n ) [ Name Canvas Serial ]
            printf
        } if
        Name /RightMouseButton eq {
            end exit
        } if
    end
} loop

null seteventlogger                      % Turn off event-logger.
```

You can run this example with `psh`, click the left mouse button, and observe the event information printed to the screen. Notice that when you click the right mouse button to exit the loop, a popup menu is displayed for the canvas under the mouse; the menu is displayed because the event-logger did not affect the normal distribution of events.

This example uses the **printf** utility, which is provided by the POSTSCRIPT language extension files. The **printf** utility is similar to the standard C `printf` utility; for more information about **printf**, see Chapter 11, "Extensibility through NeWS Procedure Files."

5

Classes

An object-oriented programming scheme based on classes is provided with the server. The code that implements the basic class mechanism is located in the `class.ps` file (see Chapter 11, "Extensibility through NeWS Procedure Files," for information about the POSTSCRIPT language files). Classes are especially useful for creating user interface components such as windows, menus, and scrollbars.

The NeWS class system is extremely flexible. You can define your own classes to build any user interface components you desire. You can also use the predefined classes that are supplied with the NeWS toolkit. The classes in the NeWS toolkit implement the OPEN LOOK™ user interface.[2]

This chapter provides an introduction to the NeWS class system; it explains how to use the operators and methods that reside in the `class.ps` file. Alphabetical lists of the operators and methods are provided at the end of the chapter as a quick reference to their syntax. You should read this chapter if you want to create your own classes, use the NeWS input classes, or use the NeWS toolkit. The toolkit classes use the basic class mechanisms described here. For a description of the NeWS toolkit classes, see the NeWS toolkit documentation.

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

5.1. Basic Terms and Concepts

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

Classes and Instances

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

[2] OPEN LOOK is a trademark of AT&T.

A *class* is a template for a set of similar objects; the objects described by the class are known as *instances* of the class. An instance of a class *inherits* the characteristics of its class but can selectively alter some of these characteristics. Classes and instances of classes are all objects; they are all represented by POSTSCRIPT language dictionaries that store the object's variables and procedures.

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

Instance Variables, Class Variables, and Methods

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

- instance variables

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

- class variables

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

- class methods

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

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

In this analogy, the physical aspects that vary from house to house correspond to the instance variables. The physical aspects that are specified in the blueprint, and thus do not vary from house to house, correspond to the class variables. The blueprint also specifes certain functions that all the houses must perform. For

example, each house must provide a working electrical system, plumbing system, and heating system. These functions specified in the blueprint correspond to the class methods. The "messages" that someone must send to invoke these functions of a house are flipping on an a light switch, turning on a faucet, and turning up the thermostat.

Inheritance and the Class Tree

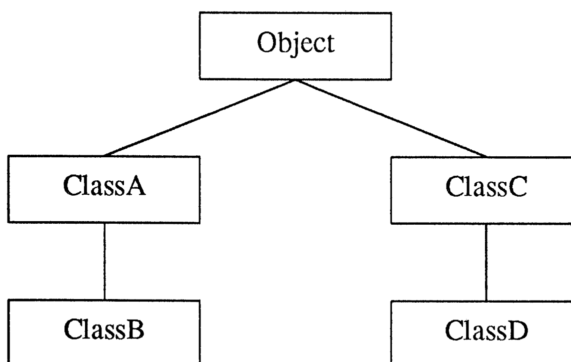
The classes in the NeWS class system belong to a class tree. The class tree is a hierarchy that is similar to, but completely separate from, the canvas tree. The root of the class tree is class **Object**. The server provides the implementation of class **Object** (in the `class.ps` file), and the other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

Superclasses and Subclasses

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

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

Figure 5-1 *A simple class tree*



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

The Immediate Superclass

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

Inheritance

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

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

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

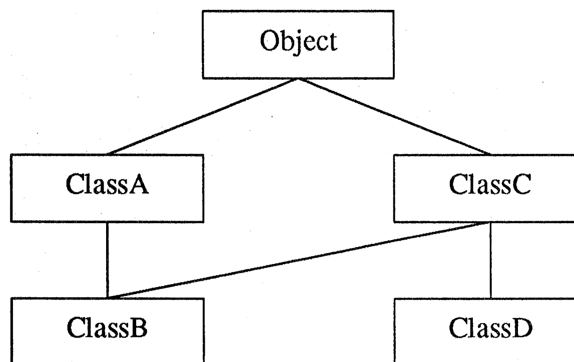
An instance can override anything that it inherits, although it usually should not override a class variable or method. An instance often changes the values associated with the instance variables that it inherits. In unusual cases, an instance can even define new variables and methods.

Single Inheritance and Multiple Inheritance

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

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

Figure 5-2 *A class tree with multiple inheritance*



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

ClassB inherits from all three of its superclasses. But a question arises: should ClassA override ClassC or vice versa? This issue is discussed in detail in Section 5.12, "Multiple Inheritance."

The Inheritance Array

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

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

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

[ClassC Object]

and the inheritance array of ClassA is

[Object]

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

[ClassA ClassC Object]

[ClassC ClassA Object]

You can choose either one of these arrays for ClassB. Section 5.12, "Multiple Inheritance," explains the details of inheritance arrays for the multiple inheritance case.

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

[ClassD ClassA Object]

and the inheritance array of an instance of ClassA is

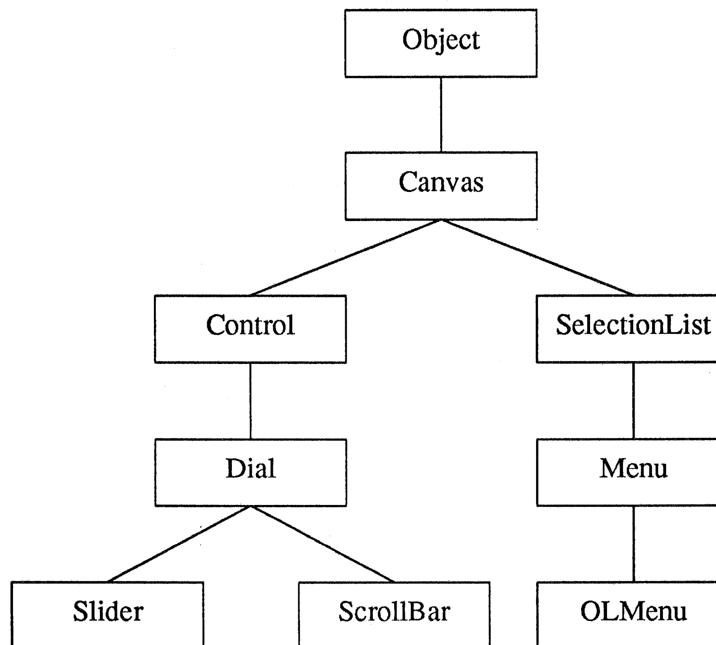
[ClassA Object]

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

A Single Inheritance Example

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

Figure 5-3 *A single inheritance example*



In this example, class **Object** has one immediate subclass named class Canvas. Class Canvas and its subclasses implement different kinds of canvases, such as menus and scrollbars.

Note that the class tree should not be confused with the canvas tree. Instances of class Canvas (and of its subclasses) represent NeWS canvas objects that exist in the canvas tree. But the instances inherit their variables and methods from class Canvas in the class tree. This dual identity is possible because NeWS magic dictionaries can have keys added to them. A NeWS canvas dictionary can be turned into an instance dictionary by adding the required instance keys. The resulting dictionary represents both a canvas object and an instance object. This arrangement is discussed in more detail in the description of the **/newmagic** method in Section 5.6, "Creating a New Instance."

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

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

Class SelectionList is a subclass of class Canvas that manages a list of items, as well as any sublists the items have; this class provides the basic operations

needed by menus. Class Menu implements a basic menu, using the operations defined in SelectionList. Class OLMenu is used to create menus with the OPEN LOOK user interface.

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

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

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

[Dial Control Canvas Object]

and class Menu's inheritance array is

[SelectionList Canvas Object]

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

[ScrollBar Dial Control Canvas Object]

and the inheritance array for MyOLMenu is

[OLMenu Menu SelectionList Canvas Object]

Summary of Terms

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

Table 5-1 *Summary of Terms*

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

5.2. Creating a New Class

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

The Class Definition

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

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

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

classbegin

classname superclasses instancevars **classbegin** —

Creates an empty class dictionary for the new class and puts it on the dictionary stack. Defines the class' instance variables and its class name.

The **classbegin** operator takes three arguments: the classname, the immediate superclass or an array of superclasses, and the instance variables. You specify the superclass(es) as one immediate superclass (the single inheritance case) or as an array of superclasses (the multiple inheritance case). See Section 5.12, "Multiple Inheritance," for an explanation of how to specify an array of superclasses. You can specify the instance variables as an array of names or as a dictionary of key/value pairs. If you use an array of names, the variables are initialized to null; if you use a dictionary, the variables are initialized to the values specified in the dictionary.

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

classend

— **classend** classname newclass

Completes the class dictionary that was left on the dictionary stack by **classbegin**. The **classend** operator constructs the inheritance array based on the superclass(es) that you passed to **classbegin** (see Section 5.12, "Multiple Inheritance," for a discussion of the inheritance array in the multiple inheritance case). The **classend** operator also compiles the class' methods (see Section 5.5, "Method Compilation") and executes any procedures in **UserProfile** that have the same name as the class (see Section 5.8, "Overriding Class Variables With **UserProfile**"). The **classend** operator returns the name of the new class (the name that you passed to **classbegin**) and the new class dictionary.

redef

name object **redef** —

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

Initializing a New Class

If a class requires some processing before the definition of the class is complete, the convention is to put the initialization code in a **/classinit** method for the class. For example, class **Object**'s **/classinit** method starts a process that listens for obsolescence events; class **Object** then handles obsolete classes and instances as explained in Section 5.11, "Obsolete Objects in the Class System."

5.3. Sending Messages With the send Operator

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

The Usual Form of send

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

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

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

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

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

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

The Steps Involved in a **send**

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

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

   Note that you might have problems if one of your methods puts a local dictionary on the stack and never removes it from the stack. See Section 5.5, "Method Compilation," if you plan to use such a method; you may need to take special precautions to ensure that the local dictionary is handled properly.

2. The **send** operator establishes *object*'s context by putting *object* and all of the classes in *object*'s inheritance array onto the dictionary stack. The inheritance array is placed on the stack with the root-most classes toward the bottom of the stack and the leaf-most classes toward the top; *object* itself is placed on the top of the stack.

3. The server searches the dictionary stack from top to bottom for the *name* method. Because *object* and the classes in its inheritance array were placed on the dictionary stack, the server finds the first occurrence of the method in *object*'s context. If the chain of classes is searched all the way back to the root without finding the specified method, an error is returned.

4. When the *name* method is found, it is executed. The arguments required by the method are taken from the operand stack, and any results of the method are put on the operand stack.

5. The initial context is then restored; the dictionary stack is restored to the state it was in before the **send** was made. If a local dictionary was removed from the top of the stack in step 1, the local dictionary is restored to its original position at the top of the stack.

The example in the following section illustrates these five steps.

Using send to Invoke a Method

This example uses the class hierarchy given in Section 5.1, "Basic Terms and Concepts." Assume that **send** is invoked as follows:

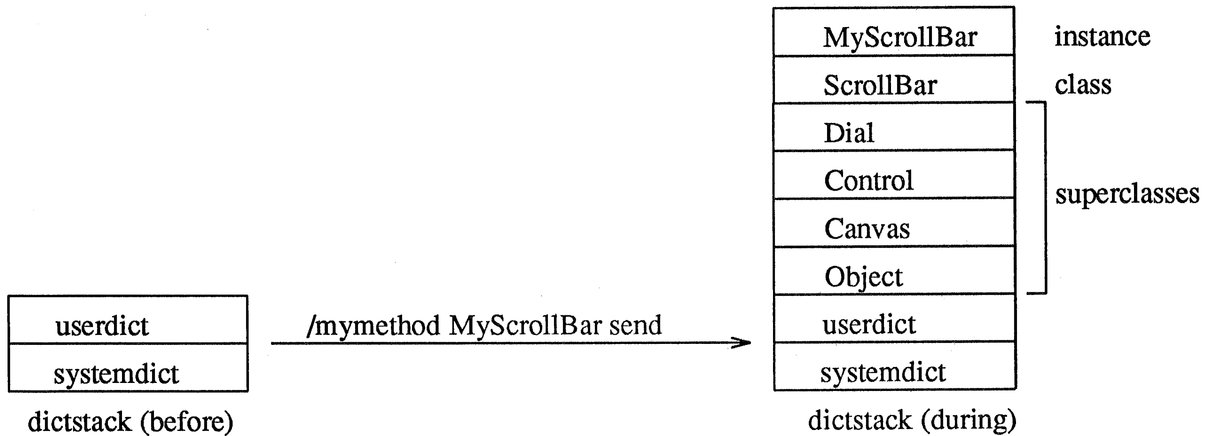
```
arg1 arg2 /mymethod MyScrollBar send
```

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

1. No existing **send** context is on the stack when this **send** is called, so nothing is removed from the stack.

2. The instance MyScrollBar and the classes in its inheritance array are pushed on the dictionary stack, as shown in the following figure:

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



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

A Nested send

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

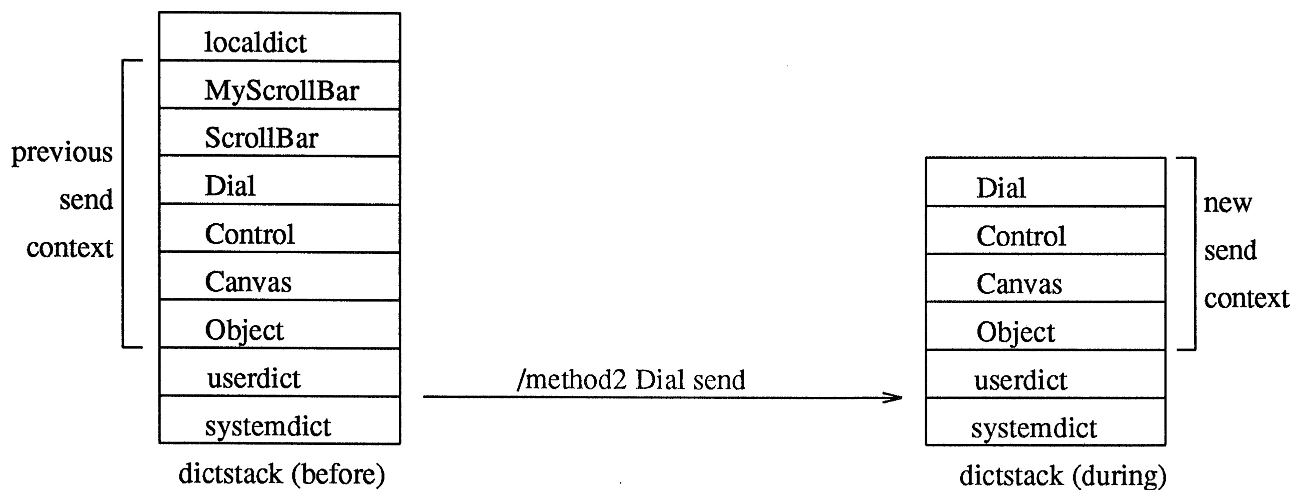
Assume that /mymethod is sent to MyScrollBar as before. The classes in MyScrollBar's inheritance array are put on the dictionary stack. Suppose that /mymethod is located in ScrollBar and that /mymethod is defined as follows:

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

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

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

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



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

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

This example is only meant to illustrate the manipulation of the dictionary stack during a nested **send**. In /mymethod, you would not actually send a message directly to ScrollBar's superclass. Instead, you would use the **super** pseudo-variable to represent the message receiver; **super** is discussed in Section 5.4, "The Pseudo-Variables **self** and **super**."

Using send to Create a New Instance

Class **Object** provides several methods for creating new instances of a class. The **/new** method is briefly introduced here; the creation of new instances is discussed in detail in Section 5.6, "Creating a New Instance."

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

```
/new MyClass send
```

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

Another Form of send

<args> procedure object **send** <results>

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

```
{procedure} object send
```

The /**doit** method must sometimes be used in conjunction with this form of send. For details, see Section 5.5, "Method Compilation."

This form of **send** bypasses the established class interface and should rarely be used. One valid use of this form of **send** is a *batch* **send**; see the /**doit** method in Section 5.5, "Method Compilation," for an example of a batch **send**.

Using send to Change the Value of an Instance Variable

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

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

You can also change the value of an instance variable by passing a new value in a procedure argument to **send** (see the subsection "Another Form of **send**," above) or by putting the value directly in the instance dictionary. Both these methods are discouraged because they ignore the established class interface and may cause problems.

For example, a class method that changes the value of an instance variable might also take a special action when the value is changed. Suppose class Dial has a /setvalue method that not only sets a dial's internal value, but also redraws the dial on the screen to reflect the new value. If you change the value of the dial without using setvalue, the dial will not be redrawn. To avoid this type of problem, you should always use an established class interface to change the value of instance variables.

Using send to Change the Value of a Class Variable

You change the value of a class variable the same way you change the value of an instance variable: you define a class method that changes the value of the variable and then invoke the method. Note that you should use **store** instead of **def** in methods that define the value of class variables. If you use **def**, you might accidently add the class variable to an instance dictionary that happens to be on top of the stack. (You can intentionally add a class variable to an instance dictionary; this action is known as *promoting* the instance variable. For details, see Section 5.9, "Promoting Class Variables to Instance Variables.")

5.4. The Pseudo-Variables self and super

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

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

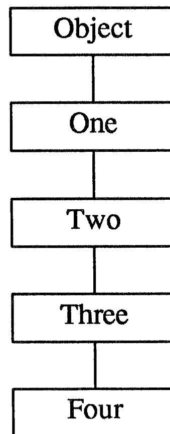
This section uses a simple example to illustrate **self** and **super**.[3] Four classes are defined as follows:

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

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

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

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

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

[3] This example is adapted from an example in Adele Goldberg's *SmallTalk—The Interactive Programming Environment*, Addison Wesley, 1984, pp 62-66.

Figure 5-6 *Class tree for* self *and* super *example*



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

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

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

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

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

the server prints to the screen is shown on the second line (in `listing` font).

The self Pseudo-Variable

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

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

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

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

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

Note that the context swapping in this nested **send** is inefficient. The same context is swapped on and off the stack several times. The NeWS class mechanism usually avoids doing these extra context swaps that occur when **self** is used with **send**; the **classend** operator *compiles* a class' methods to replace most occurrences of /method self send with a more efficient form (see Section 5.5, "Method Compilation"). Because **self** is implemented as an operator that returns an object, the construct /method self send can be executed even if it is not compiled; the compilation is done merely as an optimization.

In the next example, the /result1 message is sent to Inst2:

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

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

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

Below are four more example **sends**.

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

The super Pseudo-Variable

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

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

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

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

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

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

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

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

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

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

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

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

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

Unlike **self**, **super** is not implemented as an operator that returns an object. When the **classend** operator compiles a class' methods, each occurrence of /method super send is replaced with an operator that resolves **super** and then finds and executes the method in the current context. Thus **super** cannot be used without **send**, and it cannot be used unless the method in which it occurs is compiled. As a consequence of this implementation, the context swapping is always avoided for **sends** to **super** (see Section 5.5, "Method Compilation").

In the next example, the /result3 message is sent to Inst4:

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

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

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

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

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

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

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

Using super to Send a Message Up the Superclass Chain

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

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

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

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

Restrictions on the Use of self and super

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

Unlike **self**, **super** can only be used as an argument to **send**. The **super** pseudo-variable is not implemented as an operator that returns an object; for details on how **super** is implemented, see Section 5.5, "Method Compilation." The **super** pseudo-variable has one other restriction on its use: **super** cannot be used anywhere in a procedure passed to **send** unless the **/doit** method is used (see **/doit** in Section 5.5, "Method Compilation").

5.5. Method Compilation

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

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

Compiling self send

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

Compiling super send

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

Local Dictionaries

When a **send** is executed, any current **send** context is cleared from the dictionary stack, and the context for the message receiver is established on the dictionary stack. The **send** operator puts the message receiver on top of the dictionary stack. During execution of the method invoked by the **send,** the topmost dictionary is almost always the message receiver. However in certain cases, a method may use a *local dictionary* during its execution. A local dictionary is a dictionary that the method places on the dictionary stack while the method is executing. If a local dictionary is on the stack when a nested **send** is invoked, the local dictionary is removed from the stack before the nested method is invoked (see Section 5.3, "Sending Messages With the **send** Operator").

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

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

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

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

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

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

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

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

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

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

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

Controlling Method Compilation

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

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

/methodcompile

uncompiledproc **/methodcompile** compiledproc

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

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

/installmethod

name procedure **/installmethod** —

Creates a new method outside of a class definition. When you send **/installmethod** to an object, it installs *procedure* as a method of the object and gives the method the specified *name*. **/installmethod** compiles *procedure* by calling **/methodcompile**, and then it adds the method to the object's dictionary. The object can be a class or an instance; in the latter case, **/installmethod** creates an "instance method." An instance method is typically used to avoid creating an almost empty class that has only one method definition. (Note that **super** in an instance method resolves to the instance's class, not the superclass of the instance's class.)

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

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

/doit

<args> procedure **/doit** <results>

Compiles and executes *procedure*. The **/doit** method is used to compile a procedure that is passed to the **send** operator (see the subsection "Another Form of **send**" in Section 5.3, "Sending Messages With the **send** Operator"). You use **/doit** in the following way:

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

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

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

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

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

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

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

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

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

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

SetLocalDicts

int **SetLocalDicts** —

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

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

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

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

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

/size {
        .
        .
} def
```

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

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

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

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

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

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

In this case, the local dictionary count is 1 when the method compiler reaches /size; therefore, /size self send is not optimized. After /method1 is compiled, its contents are as follows:

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

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

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

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

After the method compiler compiles this method, its contents are as follows:

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

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

5.6. Creating a New Instance

This section discusses the methods that the `class.ps` file provides to create and initialize instances. You send the /**new** message to create a new instance of a class. A class can use the standard object creation provided by **Object**'s /**newobject** method, or the class can alter the way an object is created. For example, the /**newmagic** method can be used to create a new instance from an existing NeWS magic dictionary. A class can initialize its instances with the /**newinit** method. To request the default implementation of a class, you can send the /**newdefault** message instead of the /**new** message (/**newdefault** is discussed in Section 5.7, "Intrinsic Classes and Default Classes").

/new

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

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

```
/new MyClass send
```

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

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

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

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

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

The **/newobject** and **/newinit** methods are described in more detail below. The **/newmagic** method is also described below.

/newobject

<creationargs> **/newobject** instance

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

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

inheritance array to the instance.

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

/newinit

<initializationargs> /newinit —

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

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

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

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

%class variables:

  /ClassTime currenttime def

%methods:

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

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

  /resettime {
    /Time currenttime def
  } def

classend def
```

Class TimeKeep has a class variable, ClassTime, that is set to the time of creation of the class. TimeKeep has an instance variable named Time. Class TimeKeep's **/newinit** method first sends **/newinit** to **super**; then it calls the /resettime method to initialize the instance variable Time to be the time of creation of the instance (the time at which the method is called). The method printtime prints the value of the instance variable Time.

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

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

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

```
/printtime timer send
```

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

When you create a new instance, the **/newobject** method copies all the class' instance variables into the new instance dictionary. This copying takes less time for simple instance variables than for composite instance variables. Therefore, whenever you can avoid declaring a composite instance variable in a dictionary passed to **classbegin**, you shorten the amount of time required to create a new instance of that class. This time difference is more significant if you can arrange your class definition to avoid passing any composite instance variables to **classbegin**. To initialize a null dictionary, for example, you might define a simple instance variable to be null in the dictionary that you pass to **classbegin** and then define that variable to be a **growabledict** in a **/newinit** method for the class (see the definition of the **growabledict** utility in Chapter 11, "Extensibility through NeWS Procedure Files"). This arrangement is faster than simply defining the variable to be **nulldict** in the dictionary that you pass to **classbegin**.

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

/newmagic

<creationargs> dict **/newmagic** instance

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

The **/newmagic** method takes a magic dictionary object from the stack and uses the key/value pairs in the magic dictionary as instance variables. The instance is also given any instance variables specified by its class. The magic dictionary is turned into an instance dictionary by adding the additional instance keys; this is

possible because, by definition, a magic dictionary can have keys added to it.

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

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

    %class variables
    .
    .

    %class methods

    /newobject {
        newcanvas
        /newmagic super send
    } def

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

    .
    .
classend
```

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

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

5.7. Intrinsic Classes and Default Classes

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

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

The DefaultClass Variable

An intrinsic class must specify a *default subclass*. If the **/newdefault** message is sent to an intrinsic class, the newly created instance belongs to the intrinsic class' default subclass (see the description of **/newdefault** below).

A class' default subclass is specified by a class variable named **DefaultClass**. You can set the value of **DefaultClass** in the class definition. The example below sets the default class for Window to be MyWindow. Note that the value of the **DefaultClass** variable is the default subclass inside procedure braces; the braces are needed to defer execution until the definition of the intrinsic class is complete. (A subclass of Window is not defined until after Window itself is completely defined. And the definition of Window depends on MyWindow since MyWindow is the default subclass for Window. Therefore, a circular dependency exists. To break the dependency, the execution of **DefaultClass** is deferred by using procedure braces.)

```
/Window [Canvas]
instance variables
classbegin

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

A user can override the default implementation of a class by including a procedure in the **UserProfile** dictionary (see Section 5.8, "Overriding Class Variables With **UserProfile**").

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

/newdefault

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

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

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

```
/newdefault Window send
```

For example, if the default subclass of Window is MyWindow, the above expression causes **/new** to be sent to MyWindow. Clients written in this way are user interface independent; all that needs to be done to change to a different user interface is to change the **DefaultClass** values, which can be done easily by a user.

/defaultclass

— **/defaultclass** class

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

/SubClassResponsibility

— **/SubClassResponsibility** —

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

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

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

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

5.8. Overriding Class Variables With UserProfile

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

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

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

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

Overriding DefaultClass

A user can include a procedure in **UserProfile** that assigns a new value to a class' **DefaultClass** variable; the new value overrides the value assigned in the class definition. (For an explanation of **DefaultClass** see Section 5.7, "Intrinsic Classes and Default Classes").

For example, assume that the class definition of Window sets the default class of Window to be MyWindow. If a user wants the default implementation of class Window to be SpecialWindow instead of MyWindow, the user could add the following definition to the **UserProfile** dictionary:

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

Note that SpecialWindow must be given in braces (for details, see the explanation of **DefaultClass** in Section 5.7, "Intrinsic Classes and Default Classes").

5.9. Promoting Class Variables to Instance Variables

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

promote

name value **promote** —

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

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

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

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

unpromote

name **unpromote** —

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

```
/FrameColor unpromote
```

promoted?

name **promoted?** boolean

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

```
/FrameColor promoted?
```

Avoiding an Accidental Promotion

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

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

safe, you should always use **store** to define values of class variables.

5.10. Destroying Classes and Instances

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

/destroy

— **/destroy** —

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

/destroydependent

— **/destroydependent** —

Sent to an instance that might be shared among several instances; the **/destroy-dependent** method only destroys the instance if it is not shared. For example, several frame instances might share the same menu instance. When **/destroy** is sent to one of the frame instances, the **/destroydependent** message is sent to the instances referenced by the frame, including the menu. The menu would not be destroyed because it is still needed by the other frame instances. The **/destroy-dependent** method defaults to /destroy self send in class **Object**; classes that need **/destroydependent** should provide their own definitions.

classdestroy

class **classdestroy** —

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

/cleanoutclass

— **/cleanoutclass** —

Calls the **cleanoutdict** operator, which is a NeWS utility that undefines every key in the specified dictionary using the **undef** NeWS primitive (**cleanoutdict** is described in Chapter 11, "Extensibility through NeWS Procedure Files"). A class can override the default **/cleanoutclass** method with its own clean-up procedure, if necessary.

5.11. Obsolete Objects in the Class System

When all the references to an object are *soft*, the object is *obsolete* and the server sends an *obsolescence* event to all processes that have expressed interest in obsolescence events for that object (see Chapter 8, "Memory Management"). The processes should then remove their soft references to the object so that the server can destroy the object and reclaim the memory that it used.

When class **Object** is initialized, it starts a process named **ObsoleteEventMgr** that expresses interest in obsolescence events. When **ObsoleteEventMgr**

receives an obsolescence event from the server, it invokes a method in class **Object** that handles obsolescence events. This method performs the following actions:

- If the obsolescence event is for a class, the **classdestroy** operator is called to destroy the class (see **classdestroy** in Section 5.10, "Destroying Classes and Instances").
- If the obsolescence event is for an instance, the **/obsolete** method is sent to the instance to destroy it (see the description of **/obsolete**, below).
- If the obsolescence event is not for a class or an instance, it is simply popped from the stack.

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

/obsolete

— **/obsolete** —

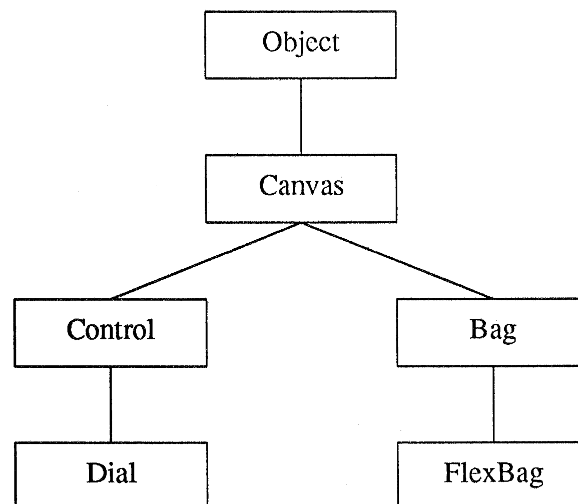
Sends **/destroy** to **self** (see the explanation of **/destroy** in Section 5.10, "Destroying Classes and Instances"). When class **Object**'s **ObsoleteEventMgr** receives an obsolescence event for an instance, the **/obsolete** message is sent to the instance to destroy it. A class rarely needs to override the default **/obsolete** method.

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

5.12. Multiple Inheritance

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

Figure 5-7 *Basic class hierarchy for the multiple inheritance examples*



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

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

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

A Simple Multiple Inheritance Example: a Utility Class

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

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

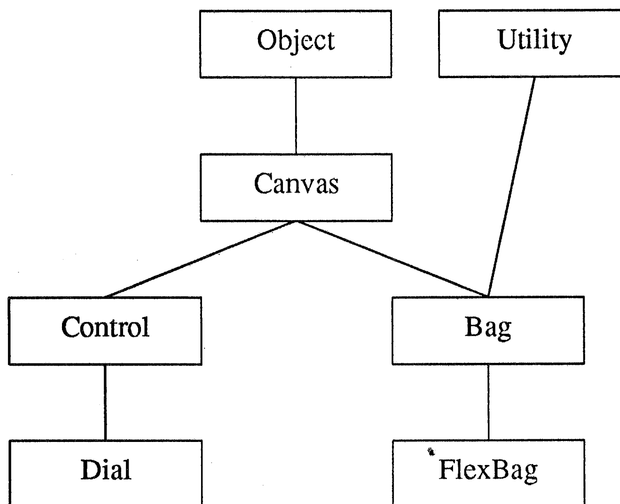
In fact, this is how class **Object** is defined. But class **Object** is the root of the class tree, whereas class Utility is a utility class. Multiple inheritance allows the classes in the main class tree to access class Utility's methods.

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

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

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

Figure 5-8 *Class hierarchy with a utility class*



A class' superclasses include the class' immediate superclasses and all of their superclasses. As shown in the diagram of the class tree, Bag's superclasses are Utility, Canvas, and **Object**. Although the tree does indicate which classes are Bag's superclasses, it does not indicate a unique order in which the superclasses should override each other. The superclasses do not belong to the same branch, so a unique leaf-to-root order is not possible.

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

[Utility Canvas Object]

[Canvas Utility Object]

[Canvas Object Utility]

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

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

If you only specify a class' immediate superclasses in the array that you pass to **classbegin**, the **classend** operator uses an algorithm to construct a default order for the inheritance array. The **classend** operator starts with a copy of the superclasses that you pass to **classbegin**, and it adds the other superclasses to build the complete inheritance array. After your new class is created, you can examine the default order of the inheritance array by sending the **/superclasses** method to the

new class. The /**superclasses** method puts the inheritance array on the operand stack (see Section 5.15, "Utilities for Inquiring About an Object's Heritage").

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

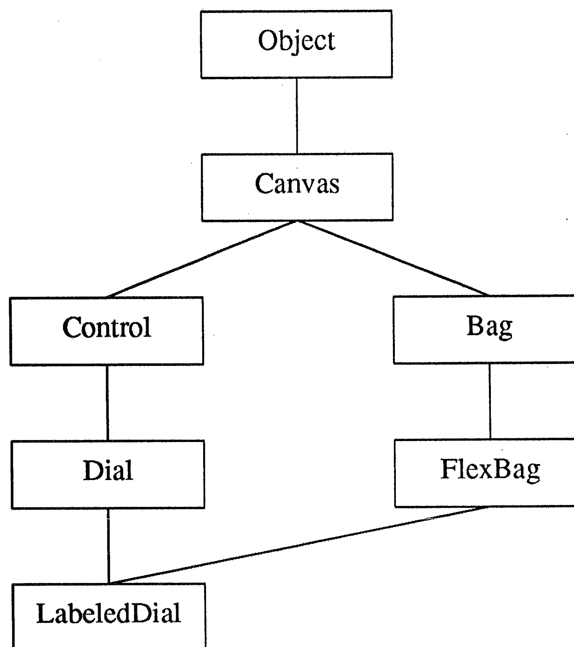
Once the inheritance array is constructed, the class mechanism works in the same way for multiple inheritance as it does for single inheritance. If a message is sent to Bag, any existing **send** context is temporarily removed, and then the classes in Bag's inheritance array are placed on the dictionary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

A More Complex Multiple Inheritance Example

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

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

Figure 5-9 *Class tree for* LabeledDial *example*



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

Rules for Valid Inheritance Array Orders

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

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

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

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

Possible Inheritance Arrays for this Example

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

Assume that the following array is given to the **classbegin** operator for LabeledDial:

[Dial FlexBag]

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

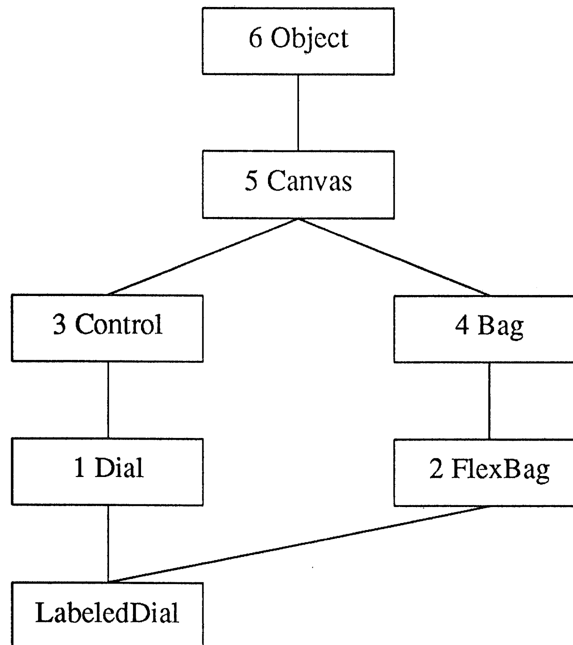
[Dial FlexBag Control Bag Canvas Object] (A)

[Dial Control FlexBag Bag Canvas Object] (B)

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

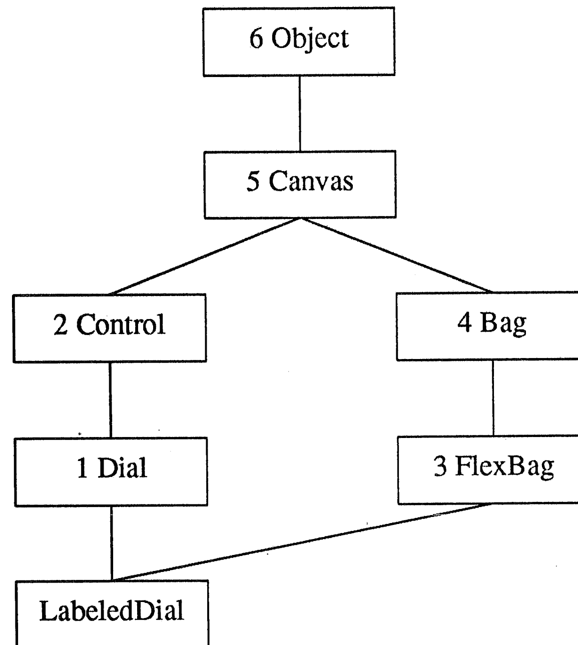
The following picture illustrates the breadth-first order (inheritance array (A) given above). Each class name has a number before it that indicates that class' position in the inheritance array.

Figure 5-10 *A breadth-first order for* LabeledDial*'s inheritance array*



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

Figure 5-11 *A depth-first order for* LabeledDial*'s inheritance array*



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

[FlexBag Dial]

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

[FlexBag Dial Bag Control Canvas Object]

[FlexBag Bag Dial Control Canvas Object]

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

Which Order Do You Choose?

You choose the order of the inheritance array based on the order in which you want the classes to override each other. If it makes no difference, you can specify just the two immediate superclasses and let the server create the default array based on your **classbegin** superclass array. To examine the default order, you can send the /**superclasses** method to the newly created class (see Section 5.15, "Utilities for Inquiring About an Object's Heritage").

Constraining the Order of the Inheritance Array

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

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

[Dial Control FlexBag]

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

[Dial Control FlexBag Bag Canvas Object]

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

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

[Dial FlexBag Control]

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

[Dial FlexBag Control Bag Canvas Object]

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

super and Multiple Inheritance

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

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

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

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

[Canvas Object]

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

Assume that LabeledDial's inheritance array is the following:

[Dial FlexBag Control Bag Canvas Object]

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

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

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

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

The class methods that set and retrieve the values of **ObjectName** and **ClassName** are described below.

/name

— **/name** name

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

/setname

name **/setname** —

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

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

/named?

— **/named?** boolean

Returns **true** if the object that receives the **/named?** message has an **ObjectName** variable. Thus, **/named?** can be sent to an instance to determine whether **ObjectName** has been promoted.

/classname

— **/classname** name

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

5.14. Utilities for Inquiring About an Object's Status

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

isobject?

object **isobject?** boolean

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

isclass?

object **isclass?** boolean

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

isinstance?

object **isinstance?** boolean

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

5.15. Utilities for Inquiring About an Object's Heritage

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

/superclasses

— **/superclasses** array

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

The following code fragment prints the names of all the classes in MyClass's inheritance array:

```
/superclasses MyClass send
[ { /classname exch send } forall ] ==
```

/subclasses

— **/subclasses** array

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

/instanceof?

object **/instanceof?** boolean

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

/descendantof?

object **/descendantof?** boolean

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

/understands?

name **/understands?** boolean

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

/class

— **/class** class

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

5.16. Utilities for Finding Objects on the send Stack

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

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

/topmostinstance

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

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

/topmostdescendant

— **/topmostdescendant** object *or* null

When the **/topmostdescendant** message is sent to a class, the method finds and returns the class' topmost descendant on the **send** stack; if no such object exists, the method returns **null**. A class' descendant is defined as an object that has that class in its inheritance array.

/sendtopmost <args> name **/sendtopmost** <results>

When **/sendtopmost** is sent to a class, it **sends** the *name* message to the topmost descendant of the class (see the explanation of **/topmostdescendant**, above). If *name* requires arguments, they should be specified.

5.17. Syntax Summary for Class Operators

classname superclasses instvars	**classbegin**	–
class	**classdestroy**	–
–	**classend**	classname newclass
object	**isobject?**	boolean
object	**isclass?**	boolean
object	**isinstance?**	boolean
name object	**promote**	–
name	**promoted?**	boolean
name object	**redef**	–
name	**unpromote**	–
–	**self**	object
<args> name object	**send**	<results>
<args> procedure object	**send**	<results>

5.18. Syntax Summary for Class Methods

–	**/class**	class
–	**/classname**	name
–	**/cleanoutclass**	–
–	**/defaultclass**	class
object	**/descendantof?**	boolean
–	**/destroy**	–
–	**/destroydependent**	–
<args> procedure	**/doit**	<results>
name procedure	**/installmethod**	–
object	**/instanceof?**	boolean
uncompiledproc	**/methodcompile**	compiledproc
–	**/named?**	boolean
<args>	**/new**	instance
<args>	**/newdefault**	instance
<args>	**/newinit**	–
<args> dict	**/newmagic**	instance
<args>	**/newobject**	instance
–	**/obsolete**	–
<args> name	**/sendtopmost**	<results>
–	**/superclasses**	array
–	**/subclasses**	array
–	**/topmostdescendant**	null *or* object
–	**/topmostinstance**	null *or* object
name	**/understands?**	boolean
name	**/setname**	–

– /SubClassResponsibility –

6

C Client Interface

A client program typically consists of two main sections: one section that performs the application's basic computations and one section that provides the application's windows or graphics. The computational part of the program should be executed in the client process; it can be written in C, FORTRAN, or any other language. The graphical part of the program is interpreted by the server process; it must be written in the POSTSCRIPT language. The POSTSCRIPT language section of the client program can be detached, sent to the server, and executed remotely with function calls.

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

Because most programmers are likely to use C as the language of the client application, an interface facility for C clients is provided with the server. The C to POSTSCRIPT facility, known as CPS, provides an interface that allows C client applications to communicate with the server. The CPS facility allows a client program to define POSTSCRIPT language routines and associate them with C function names; these functions can then be downloaded into the server and executed with function calls. Data can be passed to and from the server as arguments to the functions. The CPS facility also provides functions that open and close server communication, utilities that implement commonly used POSTSCRIPT language operators, utilities that define compressed tokens, and utilities that allow the client to read data directly from the input connection file. This chapter discusses all these aspects of the CPS facility.

Programmers can create their own interface facilities for use with other languages. The last section of this chapter gives some hints for constructing a CPS-like facility to support a different language.

NOTE *Users who wish to download pure POSTSCRIPT language programs to the server should use the* `psh` *utility (see the* `psh` *manual page in the X11/NeWS Server Guide).*

6.1. The Three Parts of a CPS Client

To use the CPS facility, a client application must have three files: a .cps file, a .c file, and a .h file. You create the .cps and .c files, then use a program called CPS to generate the .h file from the .cps file. The following three subsections describe the contents of these files, as well as the methods for creating and compiling them.

Creating the .cps File

The .cps file contains POSTSCRIPT language procedures to be executed within the server. You specify each procedure using the CPS cdef command, which associates the procedure with a C function name that the client can use to invoke the procedure. The cdef command syntax is described in detail in Section 6.3, "The cdef Command."

The comment convention for the .cps file is the same as the POSTSCRIPT language comment convention: everything from a % sign to the end of a line is a comment.

Creating the .h File

The .h file contains the POSTSCRIPT language expressions from the .cps file as macros that are comprehensible to the C compiler.

To create the .h file, you give the existing .cps file as an argument to the cps command. The following example creates a header file named myfile.h, which can then be included in the associated .c file.

```
% cps myfile.cps
```

For more information on the cps command and its options, see the cps manual page in the *X11/NeWS Server Guide*.

The .h file automatically includes the file <NeWS/psmacros.h>, which contains definitions of standard CPS macros and declarations of CPS functions that reside in libcps.a. The file <NeWS/psmacros.h> contains #include statements for both the standard I/O package <stdio.h> and the NeWS I/O package <NeWS/psio.h>.

Creating and Compiling the .c File

The .c file typically contains the main section of the C client program. In order for the .c file to reference the C function names defined in the .cps file, the .c file must contain an #include statement that includes the previously created .h file. When this file is included, any of the functions and utilities provided by CPS can also be used in the .c file.

When linking the program, you must use -lcps on the command line to add the CPS library, libcps.a, to the list of libraries searched by the linker. You must also inform the compiler and linker of the pathnames of the libraries and include files, using the cc options -I and -L. Thus, the compilation command takes the following form:

```
% cc -I$OPENWINHOME/include myfile.c -L$OPENWINHOME/lib -lcps
```

In this example, the pathnames provided to the compiler are the full pathnames of the CPS library and header files.

Including Other Header Files in the .c and .cps Files

You can include a file in the `.cps` file with the standard C `#include` syntax. You can also use the special CPS syntax `C: #include` in the `.cps` file. The resulting `.h` file produced by the CPS program will automatically include the specified header file when included by `.c` files. For example, the following CPS code will cause CPS to include `headernow.h` while converting the `.cps` file to a `.h` file. When the resulting `.h` file is included by a `.c` file, it will include `headerlater.h`:

```
#include "headernow.h"
C: #include "headerlater.h"
```

Note that the file `header.h` is just an additional header file associated with the client; it is not the CPS header file, which would be named `myfile.h` in this example. For an example of how you might want to use the `C: #include` syntax, see Section 6.8, "An Example CPS Client: The Lunar Lander Game."

The `C:` facility merely copies the rest of the line, unmodified, to the resulting `.h` file. Therefore, you can use it with `define` as well as `include`.

6.2. CPS Connection Utilities

The CPS facility provides functions for opening and closing communication with the server and for flushing the client's output buffer. These three functions form a subset of the CPS utilities automatically included by the `.h` file when the `cps` program is used. The utilities can be used in the C client provided that the CPS library is included when the client program is compiled. For a description of how to compile the `.c` file, see the subsection "Creating and Compiling the `.c` File," above.

Establishing a Connection

The following CPS function opens a connection to the server:

- `ps_open_PostScript()`

  Establishes a connection to the X11/NeWS server specified by the `NEWS-SERVER` environment variable; if the `NEWSSERVER` variable is not defined, the function establishes a connection to the local server on port 2000.

  The `ps_open_PostScript()` function opens a socket connection to the server, representing the socket as a pair of streams. Two `PSFILE` pointers, `PostScript` and `PostScriptInput`, are the conduits through which information flows between the X11/NeWS server and the client program. When the client writes to the X11/NeWS server, it writes to the file represented by the pointer `PostScript`. When the client reads information sent from the server, it reads from the file represented by the pointer `PostScrip-tInput`. All operations on these `PSFILE` pointers are performed using the `psio` package rather than the standard I/O package.

  If a connection to the X11/NeWS server is successfully established, `ps_open_PostScript()` returns a `PSFILE` pointer; if a connection is

not established, it returns 0. The `ps_open_PostScript()` function must be called before calling any procedure that needs to communicate with the server.

`PostScript` and `PostScriptInput` are two seperate streams that share the same operating system file descriptor, but `PostScriptInput` is a stream used for reading and `PostScript` is a stream used for writing. Because `PostScript` and `PostScriptInput` both use the same file descriptor, closing one stream causes the operating system file descriptor to be closed, rendering the other stream inactive.

Flushing the Output Buffer

Output from the client to the server is buffered to provide a more efficient interface mechanism. When the client calls a function that blocks while waiting for input, the contents of the buffer are automatically sent to the server. However, the client can send the contents of the buffer to the server at any time by calling the following function:

- `ps_flush_PostScript()`

  Sends the contents of the client's output buffer to the server. The function returns -1 if an error occurs or 0 if no error occurs.

Closing the Connection

The following function should be called before the client program exits:

- `ps_close_PostScript()`

  Closes the connection to the server. The function returns -1 if an error occurs or 0 if no error occurs.

Connection Example

The following example illustrates how `ps_open_PostScript()` and `ps_close_PostScript()` are used in the `.c` file.

```
/* Rough outline of a CPS client's .c file.

#include "myfile.h"

main() {

    /* Try to connect to the server.
    if (ps_open_PostScript() == 0) {

        /* Connection attempt failed.
        fprintf(stderr, "Can't connect to the server.\n");
        exit(1);
    }

    /* Successfully connected to the server.
    /* The body of the application.
    .
    .
    .

    /* The application has completed.
    /* Close the connection to the server.
    ps_close_PostScript();
}
```

6.3. The `cdef` Command

The `.cps` file must consist of `cdef` statements, each of which defines a macro that can be used by the client. A `cdef` command can define POSTSCRIPT language code to be sent to the server for execution, and it can request that the server return results after executing some POSTSCRIPT language code.

The `cdef` Syntax

The full syntax of the `cdef` command is given below:

```
cdef name(args) => tag (results)
POSTSCRIPT_code
```

The *args*, *results*, and *POSTSCRIPT_code* fields are optional; the *tag* field is required only if the *results* field is used. The symbols => are required if the *results* and/or *tag* fields are used. The parentheses around the *args* field are required regardless of whether the *args* field is used; the parentheses around the *results* field are only required if the *results* field is used. The following list describes each `cdef` field.

- *name*

  This field is the name of the macro as it appears in the client program.

- *args*

  This field represents any number of arguments to be passed to the C macro defined by the `cdef`. Each argument can be either a value to be used in the specified POSTSCRIPT language computation or a pointer into which a result is read when it is returned from the server. Note that the *args* field must come immediately after the *name* field.

  The default argument type is `int`; if a different argument type is desired, the type must be specified before the argument in the *args* list. The argument types are given in the following subsection, "CPS Argument Types."

- =>

  These symbols are used to indicate that the following integer (the *tag*) and parenthesized list (the *results*) are the specification of a packet to be received by the client when it executes this macro. Occasionally, a *tag* may be specified without any *results*; in this case, the symbols => still indicate that the following *tag* is to be received by the client.

- *tag*

  This field is an identifier typically associated with some *results*. The identifier is used to prevent confusion when multiple NeWS processes are simultaneously writing results back to the client. The identifier must be a unique integer constant or must appear in the list of arguments to the `cdef` as an integer argument.

- *results*

  This field is a list of one or more variables that receive the values returned from the server's execution of some POSTSCRIPT language code. Each variable must also be included in the *args* field, although not necessarily in the same sequence.

  Note that the =>, *tag*, and *results* fields must come together and must appear after the *name* and *args* fields; however, they can appear before, after, or in the middle of the specified *POSTSCRIPT_code*.

- *POSTSCRIPT_code*

  This field is some POSTSCRIPT language code that is invoked within the server when the *name* macro is called. The POSTSCRIPT language code can continue for several lines; indentation is not important. In the `.cps` file, one `cdef` statement is always terminated by the start of another or by an EOF.

CPS Argument Types

Each argument specified in the *args* field of the `cdef` command has an associated CPS type. The default type is `int`. To specify a different type, you must precede the argument with the appropriate type. The syntax is thus as follows:

```
cdef name(type1 arg1, type2 arg2) => tag (results) POSTSCRIPT_code
```

Most of the CPS types correspond directly to C types. The following table lists the CPS argument types. (Note that the standard C types can be sent to the server or returned from the server, but the non-C types cannot be returned from the server.)

Table 6-1 *CPS Argument Types*

CPS type	*C type*
`int`	`int`, `long`, and `char` (This is the default type in `cdef` specifications.)
`float`	`float` or `double`
`string`	`char *` strings that are null terminated
`cstring`	`char *` with an accompanying count of the number of characters in the string (Counted strings have two arguments in the C function's argument list: the first is the pointer to the string, the second is the count.)
`fixed`	a fixed-point number represented as an integer with 16 bits after the binary point
`token`	a special user-defined token used for performance improvement (see Section 6.6, "Defining User Tokens for Efficient Communication")
`postscript`	`char *` sent to the server as POSTSCRIPT language code rather than as a POSTSCRIPT language string
`cpostscript`	`char *` with an accompanying count sent to the server as POSTSCRIPT language code rather than as a POSTSCRIPT language string

The Three Types of `cdef` Macros

The `cdef` statement can be used for three purposes:

- Sending POSTSCRIPT language code to the server without requesting that values be returned.

  The POSTSCRIPT language code may send packets of data back to the client for retrieval by some other `cdef` statement.

- Sending POSTSCRIPT language code to the server, explicitly requesting that a given set of results be returned, and blocking until the results are returned. This type of `cdef` is known as requesting a *synchronous reply* because the server and client are synchronized.

- Sending no POSTSCRIPT language code to the server, but explicitly requesting that a given set of results be returned while continuing to run without blocking if a different set of results is returned. This type of `cdef` is known as requesting an *asynchronous reply* because the client and server are not synchronized.

These three types of `cdef` statements are described in detail in the following three sections.

Sending POSTSCRIPT Language Code without Returning Values

To create a `cdef` function that sends POSTSCRIPT language code to the server without requesting any results, you use the following syntax, which omits the =>, *tag*, and *results* fields:

```
cdef name(args) POSTSCRIPT_code
```

This type of `cdef` statement requires the name of the macro and the POSTSCRIPT language code, but the *args* field is optional. The parentheses around the *args* field are required even if no arguments are used.

The following example shows how `cdef` is used to create a CPS macro named `ps_moveto()`:

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

If the statement `ps_moveto(10,20)` is encountered in the `.c` file, the following POSTSCRIPT language code is transmitted:

```
10 20 moveto
```

Macros should be structured to minimize the amount of traffic that occurs between client and server. For example, you can use an initialization `cdef` statement to define POSTSCRIPT language routines that can themselves be called by subsequent `cdef` statements. The following example illustrates this point:

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

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

Invoking `ps_initialize()` transmits the definition of the POSTSCRIPT language function draw-dot a single time. Invocations of the routine `ps_draw_dot()` from the C code — for example, `ps_draw_dot(30,50)` — require the transmission of fewer bytes than would be necessary if all the POSTSCRIPT language code were transmitted each time a dot was drawn.

Receiving Synchronous Replies

To create a `cdef` function that synchronously returns results from the server, the double symbol => must be used, followed by a *tag*, a *results* field, and some POSTSCRIPT language code. Each argument in the *results* field must be specified in the *args* field; each *results* argument will contain a value returned by the server's computation.

In this type of `cdef`, the server must be made to return the *tag* and *results* values after its execution of the POSTSCRIPT language code. The NeWS language

provides two operators, **tagprint** and **typedprint**, that assist in sending tagged replies from the server to the client. The **tagprint** and **typedprint** operators are described below.

n **tagprint** –
n file **tagprint** –

Prints the integer *n* (where $-2^{15} \leq n < 2^{15}$) encoded as a tag on the specified output *file*; if no *file* is specified, prints *n* on the current output stream. The tag can then be read from the client's input connection file or returned in the *tag* field of a `cdef`.

object **typedprint** –
object file **typedprint** –

Prints *object* in an encoded form on the specified output file; if no *file* is specified, prints *object* on the current output stream. The object can then be read from the client's input connection file or returned in the *results* field of a `cdef`. The *object* can only be a number or a string.

The **tagprint** operator sends the tag to the client's input connection file, and the **typedprint** operator sends a result. By calling **tagprint** just before **typedprint** in a `cdef`, the tag and results can be placed together in the input connection file for retrieval by the client. A tag thus separates its `cdef`'s packet from any other packets that might appear in the data stream flowing from the server to the client. Before calling a `cdef` that returns a reply, the client must define the `cdef`'s tag to have some unique integer value.

The following is a generic syntax for requesting a synchronous tagged reply from the server (note that the object arguments to **typedprint** do not appear in the code example because they are left on the operand stack by the POSTSCRIPT language code):

```
#define tag tagint
cdef name(args) => tag (results) POSTSCRIPT_code
tag tagprint
typedprint
typedprint
.
.
.
```

The *args* and *results* fields are optional for a synchronous reply (and therefore the **typedprint** calls are also optional). The parentheses around the *args* and *results* are mandatory if these fields are used; the parentheses around the *args* field are required even if no arguments are used.

For a synchronous reply, when this form of `cdef` is called from the C code:

- The *PostScript_code* is transmitted to the server and executed there.

- The client blocks, waiting for the server's reply.
- The **tagprint** call sends the value of *tag* (this value being *tagint*) back to the client.
- Each **typedprint** call sends the value of one argument in the *results* field back to the client; **typedprint** should be called once for each argument specified in the *results* field. The returned values can be accessed within the `.c` file according to their variable names, as specified in both the *args* and *results* fields.

Note that the server does not force packets to begin with a tag and to contain typed data; this structure must be ensured by the client's POSTSCRIPT language code. The client should not pause in the middle of sending a tagged reply; if it does, the packet may be confused with a packet simultaneously returned as an asynchronous reply (asynchronous replies are described in the next subsection).

The following example demonstrates how to receive a synchronous reply by using a tagged *results* field. This `cdef` macro defines a C function, named `ps_bbox()`, that takes as arguments four pointers to integers. The function sets the integers to the bounding box of the current clipping path.

```
#define BBOX_TAG 57
cdef ps_bbox(x0,y0,x1,y1) => BBOX_TAG (y1, x1, y0, x0)
     clippath pathbbox        % Find the bounding box of the
                              % current clip.
     BBOX_TAG tagprint        % Send back the tag.
     typedprint               % Send back the results.
     typedprint               % y1 is on the top of the stack,
     typedprint               % then x1. Thus, the results list
     typedprint               % is in the opposite order from
                              % the argument list.
```

When `ps_bbox()` is called, it transmits the specified POSTSCRIPT language code to the server. When executed, the clippath pathbbox call returns the bounding box of the current clipping region onto the operand stack. The **tagprint** and **typedprint** operators then send the tag and results to the C client. The **tagprint** operator sends the tag 57 to the client, and the **typedprint** operators send the coordinates of the bounding box. The C client has been waiting for the tag 57; when the tag is returned, the client can access the coordinate values in the four pointers.

Receiving Asynchronous Replies

Asynchronous replies are typically required to monitor user input. The client program enters a loop and, on each iteration, checks whether values have been returned from the server.

To create a `cdef` function that receives an asynchronous reply from the server, you simply omit the *POSTSCRIPT_code* argument from the `cdef` statement as follows:

```
#define tag tagint
cdef name(args) => tag (results)
```

The *results* and *args* fields are optional, but the parentheses are required even if the arguments are not specified.

When this form of `cdef` function is called from the C code, the client's input connection file is checked. Then the following actions are taken:

- If no input is waiting, the client blocks until some input is sent from the server.
- If input is waiting (or arrives while the client is blocked), the first input item is compared with *tag*. If the input item does not match the value of *tag*, it is left in the input connection file and the `cdef` returns 0; the client then continues execution. If the first input item does match the value of *tag*, the input item is removed from the input connection file, any *results* are read into the specified variables, and the `cdef` returns 1.

Thus, a `cdef` routine that sends no POSTSCRIPT language code to the server only blocks if no input is waiting in the client's connection file; if input is waiting, execution of the client is allowed to continue even if the returned *tag* does not match the tag specified by the `cdef`.

Note that the server must still execute **tagprint** and **typedprint** to return the *tag* and any specified *results*. However, the code that calls these operators is not supplied by the `cdef` statement; instead, it must have been sent to the server by a previous `cdef` statement that the client has executed. The code in the server is then triggered by an event, such as user input, that is external to the client.

For example, the following expression, which sends a *tag* and *result* to a client, could be executed by the server whenever a menu selection is made by the user:

```
MENU_HIT_TAG tagprint
menuindex typedprint
```

Then the following `cdef` statement could be used within a client loop to receive asynchronous menu-selection messages:

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

When the function `ps_menu_hit()` is called from the `.c` file, the client blocks until input arrives from the server. When input arrives, the tag is compared to the `cdef` tag `MENU_HIT_TAG`. If the tag values match, `ps_menu_hit()` returns 1, and the value of the *results* field (in this case, an index) is passed back in the function's *args*. If the tag values do not match, the function returns 0, and the tag remains in the input connection file to be read by another `cdef` function.

Functions such as `ps_menu_hit()` can be used to construct the basic command interpretation loops of a NeWS client program, as demonstrated in the following example:

```
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 */
        }
}
```

6.4. CPS Utilities for Retrieving Input from the Input Connection File

This section describes the CPS library functions that a client can use to examine the data in its input connection file. All these functions call I/O functions that are contained in the `psio` package. Note that the client can use the `psio` functions directly, but then the client must explicitly pass the `PSFILE` structures for the files on which to read and write. CPS simplifies the task by supplying the `PSFILE` structures for the client, using the pointers `PostScriptInput` and `PostScript`.

- `ps_check_input()`

  Checks whether the input connection file contains input. Returns 1 if the connection file contains input, 0 if it does not, or -1 if an error occurs.

- `ps_query_tag(`*tag*`)`

  Searches for *tag* in the input connection file. Returns 1 if *tag* is present, 0 if it is not present, or -1 if an error occurs.

- `ps_peek_tag(`*ptag*`)`

  Examines the tag associated with the top packet in the input connection file and returns the tag's value in the pointer *ptag*. The function leaves the tag in the connection file. It returns 1 if a tag is present, 0 if something other than a tag is present, or -1 if an error occurs. If no input is in the input connection file when `ps_peek_tag` is called, the function blocks until the server sends input to the connection file.

- `ps_read_tag(`*ptag*`)`

  This function is identical to `ps_peek_tag`, except that if a tag is found in the input connection file, `ps_read_tag` removes the tag from the file. You should only use this function if you know that the tag in the connection file has no associated data; otherwise, the associated data is stranded in the file without a tag.

- `ps_skip_input_value()`

  Removes the top entry from the input connection file, regardless of what the entry is. Returns 1 if it successfully removes something from the file, or returns -1 if an error occurs. If no input is in the connection file when `ps_skip_input_value` is called, the function blocks until the server sends input to the file. This function can be used to remove a tag from the connection file, or it can be used to restore order in the file if a tag becomes separated from its associated data.

6.5. CPS Utilities for Common POSTSCRIPT Language Operators

The following table lists the CPS utilities that implement some common POSTSCRIPT language operators. You can use these utilities without defining them on the server side. The utilities are all located in the header file `<NeWS/psmacros.h>`, which is automatically included when you use the `cps` command to create your `.h` file.

Table 6-2 *CPS Utilities for POSTSCRIPT Language Operators*

Function()	*Description*
`ps_moveto(x,y)`	x y **moveto**
`ps_rmoveto(x,y)`	x y **rmoveto**
`ps_lineto(x,y)`	x y **lineto**
`ps_rlineto(x,y)`	x y **rlineto**
`ps_closepath()`	**closepath**
`ps_arc(x,y,r,a0,a1)`	x y r a0 a1 **arc**
`ps_stroke()`	**stroke**
`ps_fill()`	**fill**
`ps_show(string s)`	s **show**
`ps_cshow(cstring s)`	s **cshow**
`ps_findfont(string font)`	font **findfont**
`ps_scalefont(n)`	n **scalefont**
`ps_setfont()`	**setfont**
`ps_gsave()`	**gsave**
`ps_grestore()`	**grestore**

6.6. Defining User Tokens for Efficient Communication

The client and server can communicate using a simple stream of ASCII characters. However, the server and CPS both understand another type of encoding that is more efficient than ASCII encoding and is more natural for certain data types such as integers, floating point numbers, and double precision numbers. This second type of encoding is a very efficient binary encoding. Each syntactic entity that is sent across the communication channel in this binary encoding is known as a *compressed token.* A compressed token is composed of digits, just as a POSTSCRIPT token is composed of ASCII characters. The server and CPS recognize ten types of compressed tokens. These ten types of compressed tokens can be categorized in three main groups, as described below.

- typed tokens

  Typed tokens are compressed tokens that encode data of a certain type. There are five types of encodings for typed tokens, corresponding to the following five data types: floating point numbers, floating point double-precision numbers, integers and fixed point numbers, short strings, and long strings.

- system tokens

  System tokens are compressed tokens that encode system-defined objects. A *system token list* is provided with the server (in the file `$OPENWINHOME/etc/NeWS/systoklst.ps`). The system token list is a list of commonly used POSTSCRIPT language operators and NeWS operator extensions. Each object in the list is associated with a system token. When the server encounters a system token while scanning POSTSCRIPT language code, it immediately looks up the token's value in the system token list and acts as if it had scanned that value from the input stream. There are two types of system token encodings; one type requires only one byte, and the other type requires two bytes.

- user tokens

  User tokens are compressed tokens that encode user-defined objects. Each server file object has a *user token list* associated with it. Thus, the client's connection file has an associated user token list that the client can use to define its own user tokens. Any type of server object (for example, a name, number, canvas, or process) can be added to the client's user token list. After an object is added to the user token list, it is available whenever the scanner encounters the corresponding user token in the server's input stream. When the server encounters a user token while scanning POSTSCRIPT lanuage code, it immediately looks up the token's value in the file's user token list and acts as if it had scanned that value from the input stream.

  The user token list is an array that is variable in length, starting empty and growing as it is used to a maximum of 65,536 entries. There are three types of user token encodings. The first 32 entries in the user token list are associated with one type of user token encoding; one byte is required to refer to any of these first 32 objects. The next 1024 entries in the user token list are associated with a second type of user token encoding; two bytes are required to refer to any of these 1024 objects. The last 64,480 entries in the user token list are associated with a third type of user token encoding; three bytes are required to refer to any of these 64,480 objects.

  The user token facility allows the client to define a user token and then refer to that token by the token's *index* into the user token list. The first entry in the user token list has an index of zero, and each successive entry has an index value that is one greater than the previous entry.

When a CPS client transmits an object that can be encoded as a typed token or a system token, the CPS facility automatically encodes that object before transmitting it. And when the server returns values with the **tagprint** and **typedprint**

operators, it encodes the objects before transmitting them. If a client wants to further increase performance, it can define user tokens; a client only needs user tokens if communication and interpretation overheads are a performance problem.

User tokens allow a client to refer to an object in the server without having to retransmit its representation each time a reference to the object is made. The client can also avoid retransmitting an object by defining the object in the client's **userdict** and then referring to it by name. However, the user token method is more efficient.

User tokens can increase performance in several ways. A user token can be transmitted in fewer bytes than the object it represents; therefore, user tokens are more efficient to transmit to the server, especially in a low bandwidth environment. User tokens may also take less time for the server to interpret. Instead of assigning a name to an object, which requires a dictionary search, the object's value can be assigned to a user token; the server can interpret the user token and look up the object in the user token list more quickly than it can interpret a name object and then find the associated value in the dictionary stack. Since fonts require a significant amount of time to create using the font operators, they are often assigned to user tokens. However, even assigning the value of a commonly used name to a user token can increase performance.

The NeWS language includes operators for manipulating the user token list, and the CPS facility includes utilities for declaring and defining user tokens. These operators and utilities are described in the next three subsections. For more information about how compressed tokens are specified in the input stream, see Appendix B, "Byte Stream Format."

NeWS Operators for Manipulating the User Token List

The following operator adds an object to the user token list:

any integer **setfileinputtoken** –
any integer file **setfileinputtoken** –

The **setfileinputtoken** operator places the object *any* in *file*'s user token list at the index location specified by *integer*. If no *file* is specified, the current file (as specified by the **currentfile** operator) is used. The *integer* must be between 0 and 65,535.

The following operator retrieves an object from the user token list:

integer **getfileinputtoken** any
integer file **getfileinputtoken** any

The **getfileinputtoken** operator returns the object associated with *integer* in *file*'s user token list. If no *file* is specified, the current file (as specified by the **currentfile** operator) is used.

To determine the index of the last non-null token in the user token list, you can use the following operator:

file **countfileinputtoken** integer

The **countfileinputtoken** operator returns the index of the last non-null user token in the specified *file*. The returned index can be used as the next slot into which a user token can be stored.

Using setfileinputtoken to Define a User Token

A client can transmit some POSTSCRIPT language code that uses the **setfileinputtoken** operator to assign a NeWS object to an index in the client's user token list. To use a token to refer to that object in a `cdef`, the client must declare an argument of type `token` in that `cdef`'s argument list. When CPS encounters that argument's name in the `cdef`'s POSTSCRIPT code, it substitutes a user token whose index is given by that argument to the `cdef`; the supplied value for the argument should be an integer between 0 and 65,535.

Example

The following code could be used in a client's `.cps` file to define a user token for Times-Roman font and a user token for Helvetica font. This code also provides `cdef` macros for setting the font.

```
cdef ps_definefonttokens (int Roman, int Helvetica)
    /Times-Roman findfont 12 scalefont
    Roman
    setfileinputtoken
    /Helvetica findfont 12 scalefont
    Helvetica
    setfileinputtoken

cdef ps_settokenfont (token afont)
    afont setfont
```

The following code in the client's `.c` file could then be used to set the font:

```
int Roman = 0;
int Helvetica = 1;

ps_definefonttokens(Roman, Helvetica);

/* Set font to Roman. */
ps_settokenfont(Roman);

/* Change font to Helvetica. */
ps_settokenfont(Helvetica);
```

Using CPS Utilities to Define a User Token

A client can use CPS utilities in its `.c` file to assign a value to a user token. When any of these utilities are used, the client must first declare the user token in the `.cps` file with the CPS `usertoken` command.

Declaring the User Token

The CPS name for a user token can be declared in the `.cps` file with the CPS `usertoken` command as follows:

```
usertoken mytoken
```

This command tells CPS to substitute a user token whenever it encounters the name `mytoken` in any POSTSCRIPT language code throughout the rest of the `.cps` file. The command causes the declaration of a C variable named `mytoken_token` that is the CPS `token` type; the variable should not be declared in the `.c` file. All that remains is to define the user token's value before transmitting any of the `cdef` code that uses it.

Defining the Token's Value

The following CPS macros can be used in the `.c` file to define the value associated with a user token (note that these macros all transmit POSTSCRIPT language code that uses the NeWS operator **setfileinputtoken** to perform the association):

- `ps_define_stack_token(`*mytoken*`)`

  Takes the value on top of the operand stack in the server and defines it as the value of the user token named *mytoken*. In future messages to the server, *mytoken* has this value.

- `ps_define_value_token(`*mytoken*`)`

  Looks up the current value of the POSTSCRIPT language name mytoken and assigns that value to the user token named *mytoken*. Future changes to the value of the POSTSCRIPT language variable mytoken, or its identity as determined by changes in the dictionary stack, have no effect on the value of the user token.

- `ps_define_word_token(`*mytoken*`)`

  Assigns the executable POSTSCRIPT language name mytoken as the value of the user token *mytoken*. When the user token is sent to the server, the name mytoken is evaluated and its value is used.

CPS also offers three macros, specifically for fonts, that can be used to add a font to the user token list and then set the current font to be the font specified by the user token. These font macros are described below:

- `ps_finddef(string` *name*`,` *index*`)`

  Takes the font specified by *name*, adds it to the user token list, and returns an integer *index* that is the font's index into the user token list.

- `ps_scaledef(token` *font*`,` *size*`,` *index*`)`

  Takes the font specified by the user token *font* and the scale specified by *size*, adds the newly scaled font to the user token list, and returns an integer *index* that is the scaled font's index into the user token list.

- `ps_usetfont (`*index*`)`

  Takes the integer *index* that is the font's index into the user token list, and sets the current font to be the font given by the user token.

Examples

The following examples demonstrate the CPS `usertoken` command and the CPS utilities for defining the value of a user token.

Assume that the following lines are part of a client's `.cps` file:

```
usertoken Roman
usertoken Helvetica
cdef ps_getRoman() /Times-Roman findfont 12 scalefont
cdef ps_setRoman() Roman setfont
cdef ps_getHelvetica() /Helvetica findfont 12 scalefont
cdef ps_setHelvetica() Helvetica setfont
```

Then the current font could be set to Times-Roman or Helvetica with the following lines in the `.c` file:

```
/* Define Roman and Helvetica tokens. */
ps_getRoman();
ps_define_stack_token(Roman);
ps_getHelvetica();
ps_define_stack_token(Helvetica);

/* Set font to Roman. */
ps_setRoman();

/* Change font to Helvetica. */
ps_setHelvetica();
```

The `ps_getRoman()` macro puts Times-Roman on the top of the operand stack and scales it to be 12 points. The `ps_define_stack_token()` macro then sets the value of the user token `Roman` to be the 12 point Times-Roman font dictionary that is on top of the operand stack, and it sets the C variable `Roman_token` to the next index available in the token list. (Note that **countfileinputtoken** is not used to determine this next available index. Instead, a cached value in the POSTSCRIPT `PSFILE` structure is used. This value may not be appropriate if the client downloaded code that previously used **setfileinputtoken** to create one or more tokens.) The `ps_setRoman()` macro can then be called to set the current font to be `Roman`, which is the 12 point Times-Roman font. The equivalent macros are provided for Helvetica font.

Below is an example of how to use the special font macros to accomplish the same task. No code is needed in the .cps file, but the .c file would include the following lines:

```
/* Declare variables. */
int Roman, Roman12;
int Helvetica, Helvetica12;

/* Define Roman tokens. */
ps_finddef("Times-Roman", Roman);
ps_scaledef(Roman, 12, Roman12);

/* Define Helvetica tokens. */
ps_finddef("Helvetica", Helvetica);
ps_scaledef(Helvetica, 12, Helvetica12);

/* Set font to Roman. */
ps_usetfont(Roman12);

/* Change font to Helvetica. */
ps_usetfont(Helvetica12);
```

The first two lines declare integer variables to hold the indices for the font tokens. Then the user token for Roman font is defined. First, the ps_finddef macro is used to find the Times-Roman font and associate it with the user token Roman. The next line calls ps_scaledef to create a token for Times-Roman 12 point font; the ps_scaledef macro returns the user token's index in the variable Roman12. The equivalent two lines are given for Helvetica font. The ps_usetfont macro can then be used to look up the index Roman12 or Helvetica12 in the user token list and set the current font to be the font associated with that user token.

6.7. Debugging CPS Clients

You can test your application's POSTSCRIPT language code by typing it into an interactive psh session with the server. However, you may reach a point at which the code only works in the context of the client side of the program. Typically, a CPS program downloads a large amount of POSTSCRIPT language code in its "ps_initialize()" cdef function. You can place this portion of the POSTSCRIPT language code in a separate file and then change the initialization file to resemble the following:

```
cdef  ps_initialize()
      (work/testinit.ps) LoadFile
      ... any other initialization required
```

You can now make changes to the POSTSCRIPT language initialization code in testinit.ps (for example, adding "console (*debugging-statement* fprintf)" in certain places) without having to recompile the C side.

6.8. An Example CPS Client: The Lunar Lander Game

This section presents an example of a CPS client. The example is a game known as *lunar lander*. This section first describes how you play the game, and then it presents and explains the code that implements the game.

When you start the game, you will see a night sky with stars. You will also see a lunar lander descending toward the topography of the lunar surface. Each time you play the game, the topography will be different. Your job is to use the mouse to land the spacecraft safely on the lunar surface. If you land too quickly or at too much of an angle, you will crash. If you successfully land, you will see your astronaut exit from the lunar lander.

The lunar lander can only accelerate in a direction parallel to its vertical axis. You can use the mouse to control the lander's angle and thus its direction of thrust. You can also use the mouse to change the amount of thrust from the lander's engines, thus changing the magnitude and sense of the lander's acceleration along the direction of its vertical axis. The lander's flame always points away from the direction in which it is accelerating.

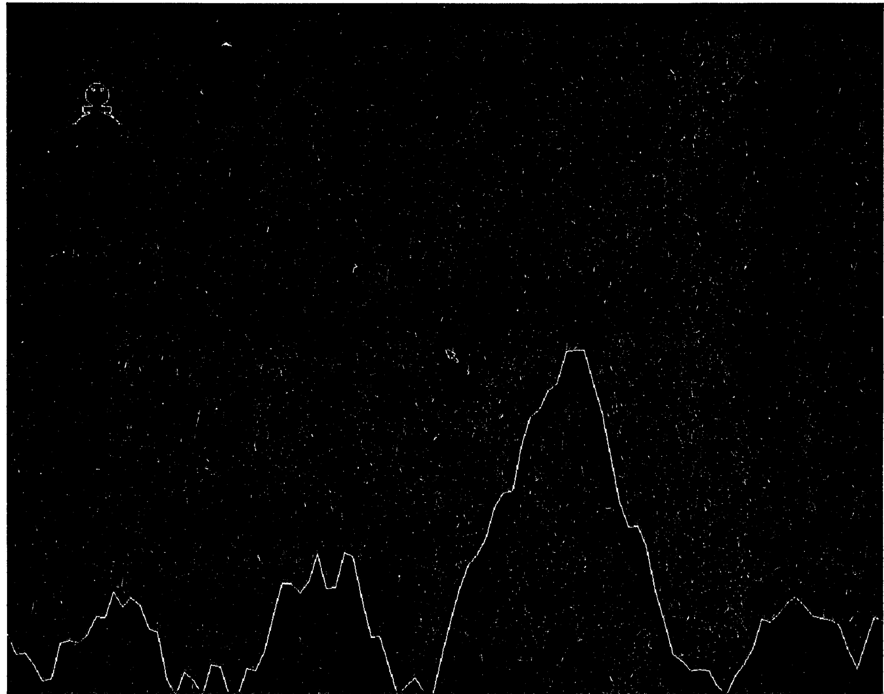
To control the angle of the lunar lander, you move the mouse horizontally to the left or right. When the cursor is in the middle of the screen, the lander is vertical. When the cursor is at the screen's left edge, the lander has its maximum clockwise rotation; when the cursor is at the screen's right edge, the lander has its maximum counter-clockwise rotation.

To control the lunar lander's engines, you move the mouse vertically up or down. When the cursor is at the screen's top edge, the lander has maximum forward thrust; when the cursor is at the screen's bottom edge, the lander has maximum backward thrust. When the cursor is in the middle of the screen, the lander has zero engine thrust, but it still descends because of the moon's gravitational acceleration. You will not see an immediate change in direction of the lander when you move the mouse because you are controlling the lander's acceleration, not its velocity.

The lunar lander can move off the top of the screen. You can move the mouse to bring it back into view. The lunar lander cannot move off the left or right edges of the screen; it bounces off these edges and flies in the opposite direction.

The figure below shows a typical lunar lander scene when the game begins:

Figure 6-1 *The lunar lander game*



You might want to try the game a few times to see how it works. The game is called `$OPENWINHOME/demo/lunar`.

Splitting the Code Between Client and Server

The computational part of a CPS program should reside on the client side, and the drawing part of the program should reside on the server side. Your application will perform much better if you adhere to this basic split in functionality because the POSTSCRIPT language is not suitable for heavy computations.

In the lunar lander example, the C side calculates the lunar lander's angle, acceleration, velocity, and position. The C side also provides the main control structure for the program. The server side handles all the drawing and the user input.

The lunar lander code is divided into three sections: a `lunar.c` file, a `lunar.cps` file, and a `lunartags.h` file. The `lunartags.h` file contains definitions of the CPS tags. The contents of these three files are given in the next three subsections (you can find the source code in `$OPENWINHOME/share/src/xnews/client/lunar/*`. The program is explained in more detail in the "Program Overview" subsection, which follows the code listings.

The Lunar Lander `.c` File

The `.c` file for the lunar lander example is given on the following pages.

```
/* Include standard math library and lunar header file. */
#include <math.h>
#include "lunar.h"
/* Define gravitational constant and number of stars to draw. */
```

```
#define GRAVITY         0.1
#define STARS           5000
/* Define array to hold y coordinate positions of terrain topography. Array has 100 slots. */
#define TERRAINSIZE     100
int     terrain[TERRAINSIZE];
/* Define slot for a particular x value and x position for a particular slot. */
#define slot(x) (((x) - xmin) * TERRAINSIZE / (xmax - xmin))
#define xpos(s) ((s) * (xmax - xmin) / (TERRAINSIZE - 1) + xmin)
/* Convert degrees to radians and back. */
#define torad(deg)      ((deg) * M_PI / 180.0)
#define todeg(rad)      ((int) ((rad) * 180.0 / M_PI))

int     xmin, ymin, xmax, ymax;

main()
{
    int ret, tag, done = 0;
    int terrainslot, randseed;
    int thrust = 0, angle = 0;
    float       shipx, shipy, shipdx, shipdy;
    float       ground, realslot;
    int crashed, finished = 0;

    /* Try to connect to the server. */
    if (ps_open_PostScript() == 0) {
        psio_fprintf(psio_stderr, "Could not connect to server.0);
        exit(1);
    }

    /* Call initialization cdef. */
    ps_initialize(&xmin, &ymin, &xmax, &ymax, &randseed);
    /* Seed the random number generator. */
    srand(randseed);
    /* Initialize the position of the lander. */
    shipx = (xmin * 9 + xmax) / 10.0;
    shipy = (ymin + ymax * 9) / 10.0;
    /* Initialize derivatives (speed of ship in x and y, which is the distance moved per timer event). */
    shipdx = 0.1;
    shipdy = -0.1;
    /* Initialize and draw landscape on the background canvas. */
    create_landscape();
    /* Send first timer event. */
    ps_starttimer(60);

    /* Loop while not done and while no I/O error. */
    while (!done && !psio_error(PostScriptInput)) {
        /* Peek at top tag in input connection file. Store value in "ret". */
        ret = ps_peek_tag(&tag);
        if (ret < 0) {
            psio_fprintf(psio_stderr, "Error reading from server.0);
            exit(1);
        }
        if (ret) {
```

```
switch (tag) {
/* If the tag is a damage tag, repair the damage. */
case DAMAGETAG:
    ps_get_tag(DAMAGETAG);
    repaint_landscape();
    break;
/* If the tag is a timer event and the game is not over, update the ship's x,y position, */
/* erase the overlay, and redraw the ship. Test for crash. If crashed, make shards procedure. */
case TIMERTAG:
    ps_get_tag(TIMERTAG);
    if (!finished) {
        /* Update the ship's x position. */
        shipx += shipdx;
        /* Don't allow ship to go horizontally off screen. */
        if (shipx <= xmin) {
            shipx = xmin;
            shipdx = -shipdx;
        } else if (shipx >= xmax) {
            shipx = xmax;
            shipdx = -shipdx;
        }
        /* Update the ship's y position. */
        shipy += shipdy;
        /* Update ship's velocity in x and y. */
        shipdy += cos(torad(angle)) * thrust / 30 - GRAVITY;
        shipdx -= sin(torad(angle)) * thrust / 30;
        terrainslot = (int) (realslot = slot(shipx));
        if (terrainslot > TERRAINSIZE - 2)
            terrainslot = TERRAINSIZE - 2;
        /* Determine ship's distance from ground. */
        ground = terrain[terrainslot]
            + (terrain[terrainslot + 1] - terrain[terrainslot])
                * (realslot - terrainslot);
        /* Erase overlay. */
        ps_eraseoverlay();
        /* Draw ship on overlay at new position. */
        ps_lander(thrust, angle, (int) shipx, (int) shipy);
        /* Test for crash. If crashed, call makeshards to define a shards procedure on */
        /* server side. */
        if (shipy <= ground) {
            shipy = ground;
            finished = 30;
            crashed = (abs(angle) >= 20 ||
                        fabs(shipdx) >= 3.0 ||
                        fabs(shipdy) >= 3.0);
            if (crashed)
                makeshards(shipdx, shipdy);
        }
    } else {
        /* This code is executed if game is "finished" but not "done"; that is, the ship has */
        /* landed or crashed but the shards or astronaut have not yet been drawn 30 */
        /* successive times. If ship crashed, call ps_boom to draw shards. If ship landed, */
        /* draw ship and astronaut. */
```

```
                    ps_eraseoverlay();
                    if (crashed)
                        ps_boom((int) shipx, (int) shipy,
                                (30 - finished) / 3 + 1, finished);
                    else {
                        ps_lander(0, 0, (int) shipx, (int) shipy);
                        ps_astronaut(((int)shipx) - 50 + finished, (int)shipy);
                    }
                    /* If shards or astronaut have been drawn 30 times, the game is done. */
                    if (--finished == 0)
                        done = 1;
                }
                /* Send another timer event. */
                ps_starttimer(60);
                /* Flush the output buffer. */
                ps_flush_PostScript();
                break;
            /* If tag is a mouse tag, get the mouse coordinates and update the thrust and angle. */
            case MOUSETAG: {
                int mx, my;

                ps_get_mouse(&mx, &my);
                thrust = (my - ymin) * 50 / (ymax - ymin) - 25;
                angle = (mx - xmin) * 200 / (xmax - xmin) - 100;
                break;
            }
        }
        /* If tag is not one of the above, skip it. */
        } else {
            psio_fprintf(psio_stderr, "Skipping bad data from server.0);
            ps_skip_input_value();
        }
    }
    /* Erase overlay and close connection to server. End program. */
    ps_eraseoverlay();
    ps_close_PostScript();
    exit(0);
}

/* Create the random landscape. Store in terrain array. */
create_landscape()
{
    int i, ty = (rand() >> 4) % 128;
    int slope = (rand() >> 4) % 64;
    int maxty = (ymax + ymin) / 2;

    for (i = 0; i < TERRAINSIZE; i++) {
        terrain[i] = ty;
        ty += (((rand() >> 4) % 64) - slope);
        if (ty <= 1) {
            ty = 1;
            slope = (rand() >> 4) % 16;
        } else if (ty >= maxty) {
```

```
            ty = maxty;
            slope = (rand() >> 4) % 16 + 48;
        } else if ((rand() >> 4) % 16 < 5) {
            slope = (int) sqrt((float) ((rand() >> 4) % 1024));
            slope = ((rand() & (1 << 5)) ? slope + 32 : 32 - slope);
        }
    }
}

/* Get damaged region of backdrop canvas and repaint with topography defined by create_landscape */
/* and with random stars. */
repaint_landscape()
{
    int         dmg_xmin, dmg_ymin, dmg_xmax, dmg_ymax;
    int         i, slot_start, slot_end;
    int         x, y, numstars;

    ps_damagepath(&dmg_xmin, &dmg_ymin, &dmg_xmax, &dmg_ymax);
    ps_moveto(0, terrain[0]);
    slot_start = slot(dmg_xmin) - 1;
    if (slot_start < 0)
        slot_start = 0;
    slot_end = slot(dmg_xmax) + 1;
    if (slot_end > TERRAINSIZE)
        slot_end = TERRAINSIZE;
    for (i = slot_start; i < slot_end; i++) {
        ps_lineto(xpos(i), terrain[i]);
    }
    ps_stroke();
    numstars = 0;
    dmg_xmax -= dmg_xmin;
    dmg_ymax -= dmg_ymin;
    for (i = dmg_xmax * dmg_ymax; (i -= STARS) > 0;) {
        x = (rand() >> 4) % dmg_xmax + dmg_xmin;
        y = (rand() >> 4) % dmg_ymax + dmg_ymin;
        if (y >= terrain[slot(x)]) {
            numstars++;
            ps_moveto(x, y);
            ps_lineto(x, y);
        }
    }
    if (numstars)
        ps_stroke();
    ps_initclip();
}

/* Make a shards procedure when lunar lander crashes. Use ps_startshards and ps_defshards to start */
/* and end the definition of a PostScript language procedure. Between the start and def, generate */
/* a random shard pattern. The shard pattern varies with the speed and angle of the crash. */
makeshards(shipdx, shipdy)
    double shipdx, shipdy;
{
    int         i, length, speed;
```

```
    float       theta, radius, angle;

    speed = (int) sqrt((shipdx * shipdx) + (shipdy * shipdy));
    if (speed < 20)
        speed = 20;
    angle = atan2(shipdy, shipdx) + M_PI;
    /* Code sent to server between here and ps_defshards is accumulating in an executable array. */
    ps_startshards();
    for (i = 0; i < 16; i++) {
        theta = torad((rand() >> 4) % 180 - 90) + angle;
        radius = (float) ((rand() >> 4) % speed + (speed / 2));
        length = (rand() >> 4) % (speed / 2) + (speed / 4);
        ps_gsave();
        ps_translate((int) (radius * cos(theta)), (int) (radius * sin(theta)));
        ps_dorotate(todeg(theta) + 90, (rand() >> 4) % speed - speed/2);
        ps_moveto(length, 0);
        ps_lineto(-length, 0);
        ps_stroke();
        ps_grestore();
    }
    ps_defshards();
}
```

The Lunar Lander .cps File

The .cps file for the lunar lander example is given on the following pages.

```
% Include the lunar.tags file, which defines tag constants, in both the .cps and .c files.

#include "lunartags.h"
C: #include "lunartags.h"

% Define the main "initialize" cdef macro.

cdef ps_initialize(xmin, ymin, xmax, ymax, randseed)
    true setpacking

    % Create some utility procs for calling from the C side.

    % Draw lander.
    /lander { % thrust rotation x y => –
        gsave 37.5 add translate rotate
            0 setgray
        % Body
            -18.5 -22.5 moveto -18.5 -15.0 lineto -5.5 -15.0 lineto
            -15.0 -5.5 lineto -15.0 5.5 lineto -5.5 15.0 lineto
            5.5 15.0 lineto 15.0 5.5 lineto 15.0 -5.5 lineto
            5.5 -15.0 lineto 18.5 -15.0 lineto 18.5 -22.5 lineto
            closepath
        % Flame
            -5.5 -22.5 moveto
            neg -22.5 add 0.0 exch lineto
            5.5 -22.5 lineto
```

```
        % Left leg
            -15.0 -22.5 moveto -15.0 -15.0 rlineto
            -2.5 0.0 rmoveto 5.0 0.0 rlineto
            -7.5 -22.5 moveto -22.5 -30.0 lineto
        % Right leg
            15.0 -22.5 moveto 15.0 -15.0 rlineto
            -2.5 0.0 rmoveto 5.0 0.0 rlineto
            7.5 -22.5 moveto 22.5 -30.0 lineto
        % Window
            -5.0 4.0 moveto -4.0 4.0 rlineto 5.0 0.0 rlineto
            closepath
            5.0 4.0 moveto 4.0 4.0 rlineto -5.0 0.0 rlineto
            closepath

            stroke
        grestore
} def
% Draw shards.
/boom { % index sc x y => -
        gsave translate dup scale shards stroke grestore pop
} def
% Draw astronaut.
/astronaut { % x y => -
    gsave
        translate
        -5.0 0.0 moveto 0.0 10.0 lineto 5.0 0.0 lineto
        0.0 10.0 moveto 0.0 22.5 lineto -2.5 2.5 rlineto
        2.5 2.5 rlineto 2.5 -2.5 rlineto -2.5 -2.5 rlineto
        -7.5 15.0 moveto 0.0 20.0 lineto 7.5 10.0 lineto
        stroke
    grestore
} def
% Send timer event for specified time in the future.
/timer { % delay-in-ms => -
        createevent dup begin
            /Name /Timer def
            /Canvas backdrop def
            /TimeStamp 3 -1 roll minim mul currenttime add def
        end sendevent
} def

% Create and map a backdrop canvas that is the size and shape of the framebuffer.
/backdrop framebuffer newcanvas def
backdrop /Retained false put
backdrop /Transparent false put
framebuffer setcanvas clippath backdrop reshapecanvas
backdrop /Mapped true put

% Start a process to listen for some events and inform the C side.
% If a damage event occurs, send DAMAGETAG. If a mouse drag event occurs,
% send a MOUSETAG and the x and y location of the event. If a timer event occurs,
% send a TIMERTAG. This process is a child of the connection process
% created when ps_open_PostScript is called.
```

```
{
    createevent dup begin
        /Name dictbegin
            /Damaged {DAMAGETAG tagprint} def
            /MouseDragged {
                dup begin
                    XLocation YLocation
                end
                MOUSETAG tagprint
                typedprint typedprint
            } def
            /Timer          {TIMERTAG tagprint} def
        dictend def
        /Canvas backdrop def
    end expressinterest
    {
        awaitevent pop
    } loop
} fork /ProcessName (lunar lander event manager) put

% Return backdrop extents and random number seed. Initialize cursor position.
    => SIZETAG (ymax, xmax, ymin, xmin, randseed)
usertime
clippath pathbbox
1 index 4 index add 2 div 1 index 4 index add 2 div setcursorlocation
SIZETAG tagprint typedprint typedprint typedprint typedprint typedprint

% Define more cdef macros.

% Start definition of shards procedure. The shards procedure takes
% an index (from 1 - 30) as an argument and returns nothing.
cdef ps_startshards()
    /shards { % index => –

% Perform a coordinate rotation. Used by makeshards function.
cdef ps_dorotate(theta, randnum)
    dup randnum mul theta add rotate

% End definition of shards procedure.
cdef ps_defshards()
    } def

% Draw lander at position x, y.
cdef ps_lander(thrust, angle, x, y)
    thrust angle x y lander

% Draw crash shards for iteration given by index.
cdef ps_boom(x, y, sc, index)
    index sc x y boom

% Draw astronaut at x, y.
cdef ps_astronaut(x, y)
    x y astronaut
```

```
% Call timer procedure to send timer event at currenttime plus delay.
cdef ps_starttimer(delay)
    delay timer

% Set current canvas to be the framebuffer's overlay. Initialize by erasing it.
cdef ps_eraseoverlay()
    fboverlay setcanvas erasepage

% Receive an asynchronous reply. Packet contains mouse tag and x, y coordinates of the mouse drag event.
cdef ps_get_mouse(x, y)    => MOUSETAG (y, x)

% Receive an asynchronous reply containing only a tag.
cdef ps_get_tag(tag) => tag

% Translate by x, y.
cdef ps_translate(x, y)
    x y translate

% Set backdrop canvas' clipping path to damage path. Send synchronous reply containing a damage path
% tag and the coordinates of the bounding box of the damaged region.
cdef ps_damagepath(xmin, ymin, xmax, ymax)
    backdrop setcanvas
    damagepath clipcanvas
    0 fillcanvas
    1 setgray
    clippath pathbbox
    DMGPATHTAG tagprint          => DMGPATHTAG (ymax, xmax, ymin, xmin)
    ceiling typedprint ceiling typedprint
    floor typedprint floor typedprint

% Remove any existing canvas clipping path.
cdef ps_initclip()
    newpath clipcanvas
```

The Lunar Lander `lunartags.h` File

The lunar lander code places the definitions of all its `cdef` tags into one header file named `lunartags.h`. This header file is included in both the `.cps` file and the `.c` file. The `lunartags.h` file is given below.

```
#ifdef notdef
    % This file contains the definitions of the tags that are used to
    % communicate information from the server back to the C program.
    % (Everything between ifdef and endif is a comment.)
#endif

#define SIZETAG      0
#define DAMAGETAG    1
#define MOUSETAG     2
#define TIMERTAG     3
#define DMGPATHTAG   4
```

Program Overview

The lunar lander program uses CPS connection utilities, POSTSCRIPT language utilities, input connection file utilities, and all three types of `cdef` macros (synchronous replies, asynchronous replies, and no replies). This section provides an overview of the lunar lander program, focusing on the aspects of the code that relate to the client-server interface. This discussion is organized around the `.c` file.

Set-Up and Connection to the Server

The `.c` file begins by including the standard C math package `math.h` and the CPS header file `lunar.h`. The file then defines some constants and macros. The main program begins with some variable declarations followed by a call to the CPS utility `ps_open_PostScript`.

Initialization

After a connection to the server is established, the `ps_initialize` macro is called to initialize the server side. The `ps_initialize` macro takes the following actions:

- It defines some POSTSCRIPT language procedures that can be called from other `cdef` macros.
- It creates and maps a canvas, named backdrop, that is used for the background of the lunar lander game.
- It forks a process that expresses interest in mouse drag events, damage events, and lunar lander timer events. (The lunar lander code sends timer events to tell the C side to recalculate the lunar lander's position.) The interest that is expressed has backdrop in its **Canvas** field and a dictionary in its **Name** field. The dictionary contains executable values for each of the three types of events that may be received. If a damage event is received, the **tagprint** operator sends the `DAMAGETAG` value to the client. If a mouse drag event is received, the **tagprint** and **typedprint** operators send the `MOUSETAG` value and the *x* and *y* coordinates of the event back to the client. If a timer event is received, the **tagprint** operator sends the `TIMERTAG`

value to the client.

- It synchronously returns the coordinates of **backdrop**'s bounding box and a random number (the **usertime**) to be used as a seed for the random number generator.

After calling `ps_initialize`, the C side initializes the coordinates of the ship so that the ship starts in the upper-left corner of the background canvas. The ship's *x* and *y* velocities are initialized so that the ship descends to the right.

Next, the C side calls the function `create_landscape`. This function, defined after the main program in the `.c` file, creates a random topography for the background.

To start the game, the C side then calls the `cdef` macro `ps_starttime` to send a timer event with a delay of 60 milliseconds. The `ps_starttime` macro simply calls the utility procedure **timer**, which was defined by `ps_initialize`; the **timer** procedure creates and sends an event with a **Name** of **Timer** and a **TimeStamp** of the current time plus the specified delay.

The Main Control Loop

The main control loop executes while the game is not done and while no I/O error occurs. First, the client's input connection file is checked with the CPS utility `ps_peek_tag`. The input item is returned in the variable `ret`. The following list summarizes the actions that are taken depending on the value of `ret`.

- If the returned value is less than zero, an error message is printed and the program exits.

- If the returned value is a `DAMAGETAG`, the `cdef` macro `ps_get_tag` is called to retrieve the tag from the input connection file. Then the `repaint_landscape` function is called to repaint the damaged area of the screen. The background is painted black, the topography previously created with `create_landscape` is drawn, and a random star pattern is drawn above the topographic profile. Note that when the unretained background canvas is initially mapped by the `ps_initialize` function call, the canvas receives damage and is painted by `repaint_landscape`.

- If the returned value is a `TIMERTAG`, the program removes the tag from the input connection file and then checks to see whether the ship has reached the ground (that is, whether the ship has crashed or landed).

  If the ship has not crashed or landed, the ship's coordinates and velocities in *x* and *y* are updated. The framebuffer's overlay is erased with a call to the `cdef` macro `ps_eraseoverlay`. The ship is then drawn on the overlay at its new *x* and *y* position with the `cdef` macro `ps_lander`. The program then checks to see if the new position means that the ship has crashed. If so, the code calls the `makeshards` function to create the **shards** procedure on the server side. Then another timer event is sent, and the control goes back to the beginning of the `while` loop.

  If the ship has crashed or landed, the program erases the overlay and draws either some shards or the ship with an astronaut exiting from the ship. A `finished` variable is decremented to count how many times the program

draws the shards or the astronaut. The explosion shards are drawn 30 successive times; the shards begin at the crash site and increase in size and distance each time they are drawn. The astronaut is also drawn 30 times, each time slightly farther from the ship. Unless the program has drawn the shards or astronaut 30 times, the game is not done; another timer event is sent, and the control goes back to the beginning of the `while` loop. If the program has drawn the shards or astronaut 30 times, the game is done and the `while` loop is exited.

- If the returned value is a `MOUSETAG`, the `cdef` macro `ps_get_mouse` is called to retrieve the mouse tag and the event coordinates from the client's input connection file. The ship's acceleration and angle are recalculated based on the user input from the mouse.
- If the returned value is not one of the above tags, the CPS `ps_skip_input_value` utility is called to skip that entry in the client's input connection file. This should not happen once the program is fully debugged and working.

Clean-Up

When the game is done, the overlay is erased and the CPS utility `ps_close_PostScript` is called to close the connection with the server. The program then exits.

Making the Shards: A Special Type of `cdef`

The shards are made in a special way to demonstrate that a `cdef` macro's POSTSCRIPT language code can be the start or the finish of a POSTSCRIPT language procedure definition. The `ps_startshards` macro simply begins the definition of a POSTSCRIPT language procedure named shards by placing the procedure name and an open procedure bracket in the server's input stream. The `ps_defshards` macro completes the definition by placing a closed procedure bracket and the token def in the server's input stream.

When the C side determines that the lunar lander has crashed, it calls the `makeshards` function, which is defined after the main program in the `.c` file. The `makeshards` function computes the velocity of the lander and then calls the `cdef` macro `ps_startshards` to begin the definition of the shards procedure on the server side. After calling `ps_startshards`, the `makeshards` function creates a fragment of POSTSCRIPT language code for each of the 16 shards. Each POSTSCRIPT language fragment draws one of the shards; the orientation and length of each shard are random, but they are influenced by the velocity and angle of the ship at the time of the crash. After the code fragments are transmitted, `makeshards` calls `ps_defshards` to complete the executable array and to define the shards procedure on the server side.

The shards procedure is then available to be called from the POSTSCRIPT boom procedure, which is a utility called by `ps_boom`. The C code calls `ps_boom` 30 times, as described previously.

The `makeshards` function demonstrates how to create a POSTSCRIPT language procedure from within the `.c` file, assuming that the `.cps` file already contains the necessary macros for starting and ending the POSTSCRIPT language procedure. In this case, the shards procedure is easier to build on the C side

because it is based on the results of some computations that can only be performed after the lander has crashed.

6.9. Creating an Interface for Clients Not Written in C

The POSTSCRIPT language (with NeWS extensions) and C are the only languages that are supported for NeWS clients. The support for C is provided by the CPS facility. POSTSCRIPT programs can be downloaded with the `psh` program.

If you want to create a client program in some language other than the POSTSCRIPT language or C, you must write a CPS-like program that is appropriate for the client's language.

Hints for Creating a Facility Equivalent to CPS

The basis for a CPS-like program should be the input and output facilities used by CPS (the `psio` package). The program should contain routines for calling the I/O facilities, macros that can be expanded into invocations of them, or similar features.

To provide runtime output for your CPS-equivalent program, you must create a function similar to the C function `pprintf()`, which provides the runtime output for the CPS program. This function is invoked in a manner similar to `fprintf()` (3S), taking a format string that is interpreted in a similar way. When the format strings contain `%s`, `%d`, or any of the other formatting specifiers, the corresponding arguments are transmitted as compressed binary tokens. The rest of the format string is transmitted as specified; it may contain compressed tokens or simple ASCII. See Appendix B, "Byte Stream Format," for a description of the server's byte stream format.

Input that the X11/NeWS server transmits to the client appears as bytes that can be read from the server I/O stream. The format of these bytes is specified entirely by the POSTSCRIPT language code downloaded by the client into the server; thus, it can be as simple or complex as is required. The NeWS language contains operators for writing objects back to a client using the same compressed binary format that the client uses to write to the server (see the description of **tagprint** and **typedprint** in Chapter 10, "NeWS Operator Extensions"). Thus, your package should contain functions or macros to facilitate the translation of these returned binary tokens into data that the client program can use.

Contacting the Server

To contact the server from a UNIX environment, you must obtain the correct IP address and port number of the server and connect to it. One way of obtaining the address is to examine the environment variable `NEWSSERVER`. This variable contains a string of the following form:

```
3227656822.2000;myhost
```

The number before the period is the 32-bit IP address of the server in host byte order. The number after the period is the server's IP port number. To contact the server, you must create a socket and connect it to the IP address and port specified by these numbers. The name that follows the semicolon in the `NEWSSERVER` variable is the text name of the host on which the server is running.

You can use the `newsserverstr` command to generate the appropriate string for the `NEWSSERVER` environment variable. See the `newsserverstr` manual page in the *X11/NeWS Server Guide* for details.

Once a connection has been established, you can simply write bytes down the stream, as described in Appendix B, "Byte Stream Format." Remember, you do not need to use the compressed binary tokens; they are merely an optimization. ASCII POSTSCRIPT language code can be sent without the use of compression.

7

Debugging

NeWS provides a *debugging facility* that allows you to set breakpoints and print messages to special output windows. The facility is provided as a POSTSCRIPT language extension file and can be modified by users.

This chapter describes the debugger and the operators it provides.

7.1. Loading the Debugger

The NeWS debugging facility is provided as the POSTSCRIPT language extension file `debug.ps`. Note that this file is not automatically loaded during the standard initialization process; thus, if you wish to load the debugger, you must execute the following command, either by adding it to your `.user.ps` file or by typing interactively in a **psh** window:

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

7.2. Starting the Debugger

To start the debugger, open a **psh executive** connection to the server and start the debugger with the **dbgstart** operator. This is demonstrated by the following example:

```
paper% psh
executive
Welcome to X11/NeWS Version 2.1
dbgstart
Debugger installed.
```

7.3. Using the Debugger

NeWS provides two kinds of debugging commands:

- Commands to be executed from client programs (*client commands*)
- Commands to be executed interactively by the user (*user commands*)

The **dbgstart** operator forks a debugger process that is attached to the `psh` connection and listens for debugger-related events generated by client commands. (All client commands broadcast debugger events to these debugger daemons.) Any client command that causes printing will print in each debugging `psh` connection.

Multi-Process Debugging

Since NeWS is a multi-process environment, you may often need to debug several processes at one time. The solution the debugger implements is to have each debugging connection maintain a list of processes that are paused for debugging. This list is printed via the **dbglistbreaks** command below. It is also printed whenever a new break occurs. Any of the listed breaks can be *entered* using the **dbgenterbreak** command. This swaps the `psh` debugging context out and replaces it with the paused process. The context currently consists of the dict stack and operand stack.

7.4. Client Commands

These are the client commands:

dbgbreak

name **dbgbreak** –

Causes the current client to pause, printing the pending breaks in all debugger connections. *Name* is used as a label in the list to distinguish between breaks, e.g. /Break1.

See also: **dbgbreakenter, dbgbreakexit**

dbgprintf

formatstring argarray **dbgprintf** –

Prints on each debugger connection, in **printf** style. If there are no debugger connections, it prints on the console. Thus the following code

```
(Testing: % % %\n) [1 2 3] dbgprintf
```

will print:

```
Testing: 1 2 3
```

on each debugger connection.

See also: **printf, dbgprintfenter , dbgprintfexit**

In addition to the above explicit calls to the debugger, errors cause the debugger to be implicitly invoked. This is done by the debugger putting a special error dictionary in the system dictionary. Each error slot in this debugger-supplied dictionary has a call to the debugger for each error. See the *PostScript Language Reference Manual* for details on error handling.

<errors>

– **<errors>** –

While debugging, a client error causes the client program to break to the debugger. This is *exactly* the same as inserting the code '/<errorname> dbgbreak' at the point the error occurred. Here is the result of encountering an **undefined** error while a debugger is running:

```
Break:/undefined from process(4154624, breakpoint)
Currently pending breakpoints are:
    1: /undefined called from process(4154624, breakpoint)
```

7.5. User Commands

Most of the user-level debugger commands come in two forms: one that explicitly takes a breaknumber and one that does not. The general rule is:

- A command of the form *cmdname***break** expects an explicit breaknumber for its argument.
- A command of the form *cmdname* (without "-break") uses an implicit breaknumber. This number is generally the currently entered break, or the last break in the list if there is no currently entered break.

The implicit form is primarily used in the most common case of only one break pending, or where constantly restating the breaknumber for the currently entered process would be arduous.

The user-commands are as follows:

dbgstart

– **dbgstart** –

Make the current connection to the server a debugger. Required before any of the other commands below can be used.

dbgstop

– **dbgstop** –

Removes the debugger from your `psh` connection.

dbglistbreaks

– **dbglistbreaks** –

List all the pending breakpoints resulting from **dbgbreak** and **<errors>** above. They are listed in the following form:

```
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
    3: /menubreak called from process(5177764, breakpoint)
    4: /undefined called from process(4154624, breakpoint)
```

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

dbgbreakenter

name **dbgbreakenter** –
[dict name] **dbgbreakenter** –
Modify the named procedure to call **dbgbreak** just after starting. If the top of the operand stack is an array, it should contain a dict and the name of a procedure in the dict. Thus to break when any new window is made:

```
[DefaultWindow /new] dbgbreakenter
Break:/new from process(4050350, input_wait)
Currently pending breakpoints are:
    1: /new called from process(4050350, input_wait)
```

See also: **dbgbreak**

dbgbreakexit

name **dbgbreakexit** –
[dict name] **dbgbreakexit** –
Modify the named procedure to call **dbgbreak** just before exiting.
See also: **dbgbreak**

dbgremovebreak

breaknumber **dbgremovebreak** –
dbgremovebreak looks at the top execution stack entry of the stopped process. If it is a regular (non-packed) array and its current execution point is a call to **dbgbreak**, the **dbgbreak** is replaced with pop. Once a breakpoint is removed, the procedure must be redefined in order to put the breakpoint back.

dbgremove

– **dbgremove** –
dbgremove calls **debgmovebreak** on the currently entered breakpoint.

dbgprintfenter

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

dbgprintfexit

name formatstring argarray **dbgprintfexit** –
[dict name] formatstring argarray **dbgprintfexit** –
Modify the named procedure to call **dbgprintf** with *formatstring* and *argarray* just before exiting. The effects of this change will persist until the NeWS server is restarted. Note that *argarray* can be an executable array if you want to defer evaluation of the arguments until the **dbgprintf** occurs.

```
[DefaultWindow /reshape] (resize: % % % %\n)
   {FrameX FrameY FrameWidth FrameHeight} dbgprintfexit
resize: 91 100 179 181
resize: 91 94 223 187
```

See also: **dbgprintf**

dbgwherebreak

breaknumber **dbgwherebreak** –
Prints a exec stack trace for the process identified by *breaknumber*:

```
1 dbgwherebreak
Level 1
  { /foo 10 'def' /bar 20 'def' /A 'false' 'def' /B 'true'
    'def' /msg (Hi!) 'def' (Testing: %\n) 'mark' msg ] dbgprintf
    /oneB *dbgbreak } (*21,22)
Level 0
  { 100 'dict' 'begin' array{22} *'loop' 'end' } (*4,6)
```

The asterisk indicates the currently executing primitive in each level. The two numbers following each procedure are the index, relative to zero, of the asterisk and the size of the procedure. This is useful information for using **dbgpatch**.

dbgwhere

– **dbgwhere** –
Prints the execution stack for the currently entered process or for the last process listed if no process is currently entered.

dbgcontinuebreak

breaknumber **dbgcontinuebreak** –
Continues the process identified by *breaknumber*.

dbgcontinue

– **dbgcontinue** –
Continues the currently entered process or the last process listed if no process is currently entered.

dbgenterbreak

breaknumber **dbgenterbreak** –
As far as possible, make this debug connection have the same execution environment as the process identified by *breaknumber*. Currently, this includes the operand stack and the dictionary stack. Thus **dbgenterbreak** allows you to browse around in the given process' state. If **dbglistbreaks** is executed while within an entered process, the listing will indicate that process with a "=>" in the left margin:

```
3 dbgenterbreak
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
  =>3: /menubreak called from process(5177764, breakpoint)
```

dbgenter

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

dbgexit

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

dbgcopystack

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

dbgcallbreak

arg clientproc breaknumber **dbgcallbreak** –
Execute **clientproc** in the broken process with *arg* as data. The **clientproc** primitive will be executed (in the client environment) with the *arg* on the stack, thus is responsible for popping it off.

dbgcall

arg clientproc **dbgcall** –
Implicit version of **dbgcallbreak**.

dbggetbreak

breaknumber **dbggetbreak** process
Returns the NeWS process object for the given breaknumber.

dbgpatchbreak

level index patch breaknumber **dbgpatchbreak** –
Patch the execution stack for breaknumber process. The patch overwrites the word in the executable at the given level, and at the given index within that level. Prints the resulting execution stack (**dbgwhere**).

dbgpatch

level index patch **dbgpatch** –
Patch the implicit process.

dbgmodifyproc

name/[dict name] headproc tailproc **dbgmodifyproc** –
Modify the named procedure to execute *headproc* just before calling it, and to call *tailproc* just after calling it. In affect, '{proc}' becomes '{headproc proc tailproc}.' This is the mechanism used for implementing **dbgbreakenter/exit** and **dbgprintfenter/exit**.

dbgkillbreak

breaknumber **dbgkillbreak** –
Kills a breakpointed process, removing it from the breaknumber list.

dbgkill

– **dbgkill** –
Kills the default process.

7.6. Debugging Hints

Here are some miscellaneous tips for debugging.

Using Aliases

Because the debugger is based on the POSTSCRIPT language, the above commands can easily be modified or overridden entirely. One common change is to define some easily-typed aliases for the above verbose names. The following POSTSCRIPT language code does the trick; you can add this to your `.user.ps` file to make the aliases available in all debugging connections.

```
/dbe {dbgbreakenter} def
/dbx {dbgbreakexit} def
/dc  {dbgcontinue} def
/dcb {dbgcontinuebreak} def
/dcc {dbgcopystack dbgcontinue} def
/dcs {dbgcopystack} def
/de  {dbgenter} def
/deb {dbgenterbreak} def
/dgb {dbggetbreak} def
/dk  {dbgkill} def
/dkb {dbgkillbreak} def
/dlb {dbglistbreaks} def
/dmp {dbgmodifyproc} def
/dp  {dbgpatch} def
/dpe {dbgprintfenter} def
/dpx {dbgprintfexit} def
/dw  {dbgwhere} def
/dwb {dbgwherebreak} def
/dx  {dbgexit} def
```

Using 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.

8

Memory Management

In any software system, a limit must be imposed on the number of objects that are allowed to exist; otherwise, storage requirements become too great and performance is impaired. Thus, the usefulness of existing objects should be continually monitored; those objects that cease to be useful should be destroyed and their storage reclaimed.

The server provides a facility of *reference counting* that allows objects to survive as long as *references* to them exist. A reference is created by the system whenever one object becomes associated with another; a reference is a pointer from one object to another. When all references to an object are removed, the storage occupied by the object is automatically reclaimed.

This chapter explains the principles of reference counting and discusses the operators that the server provides for memory management. The chapter also discusses strategies for debugging memory problems and describes the server's debugging tools and operators. The final section of the chapter discusses the *unused font cache*, which is a cache that the server uses to balance the memory cost of storing fonts against the performance cost of reloading fonts.

8.1. Reference Counting

The server counts references that are made to a given object and uses this count to determine how long the object is maintained in storage. Many operations that are applied to an object have the effect of adding or removing references to that object.

Counted and Uncounted Objects

For memory management purposes, two kinds of objects exist: *uncounted objects* and *counted objects*. These two types of objects are described below.

- uncounted objects

  These objects are simple resources, such as booleans, fixed numbers, and real numbers. These objects are not shared and therefore have no reference count.

- counted objects

  These objects, which include all other resources that the system contains, can be shared within the system. Thus, they are reference counted and can be systematically removed when they become useless.

The following two tables list the types of objects that are uncounted and counted, respectively:

Table 8-1 *Uncounted Object Types*

booleantype	**marktype**	**realtype**
colortype	**nametype**	**savetype**
fonttype	**nulltype**	
integertype	**operatortype**	

Table 8-2 *Counted Object Types*

arraytype	**environmenttype**	**pathtype**
canvastype	**evettype**	**processtype**
colormapentrytype	**filetype**	**stringtype**
colormaptype	**graphicsstatetype**	**visualtype**
cursortype	**monitortype**	
dicttype	**packedarraytype**	

References to Counted Objects

The server supports two kinds of references: *counted* and *uncounted*. Counted references can be either *hard* or *soft*.

Counted References

A *counted reference* affects the existence of an object. As long as an object has at least one counted reference, the object continues to exist in memory, and its storage cannot be reclaimed.

Note that a reference is always considered to be the property of the object that is referenced. Thus, if a reference points from A to B, the reference belongs to B and is included in B's reference count; the existence of the reference ensures that B remains in storage. When A is destroyed, its reference to B is destroyed, making B available for garbage collection if no other counted references point to B.

See the subsection "Reference Tallies" (below), for information on how the server counts references.

Uncounted References

An *uncounted reference*, which is created only by the server itself, never affects the existence of an object; it is not included in the reference count and is automatically cleaned up by the garbage collection procedures.

Uncounted references are used to avoid *circular references*. A circular reference occurs when two objects point to each other with counted references; neither object can be destroyed, since each continues to be referenced by the other.

To prevent circular references from occurring in the canvas hierarchy, the server ensures that a parent canvas always references its child canvas with an uncounted reference. This arrangement allows the child canvas to be destroyed and its storage immediately reclaimed when no counted references to the child remain. However, a child canvas always references its parent with a counted reference. Thus, the parent is never destroyed while any of its children exist, regardless of the removal of other references to the parent.

NOTE *The uncounted reference that points from a parent canvas to its child is used by server internals to perform access operations such as the following:*

```
NewCanvas /TopChild get
```

When this code is encountered, the server locates the top child of the specified canvas by tracing the appropriate uncounted reference.

Soft References and Obsolescence Events

Since uncounted references are created only by the server, NeWS programmers cannot use them to prevent circular references from occurring in the objects they themselves define; instead, programmers must use *soft references.* A soft reference is created with the **soften** operator (described in Section 8.2, "Memory Management Operators").

A soft reference is not an uncounted reference; it is a counted reference that ensures the continued existence of the object to which it points. However, soft references affect storage reclamation because the server associates them with *obsolescence events.*

An obsolescence event is automatically generated by the system when an object is preserved only by soft references (that is, when all the remaining references to it are soft). The event, which has **Obsolete** in its **Name** field and a copy of the obsolete object in its **Action** field, signifies that all remaining references to the object are soft.

Any process that uses the **soften** operator to soften an existing hard reference should also express interest in receiving an obsolescence event for that object. When the obsolescence event is distributed and successfully matched to the interest, the event can be passed to a handler that removes the process' soft reference. When all references have been removed, the object is automatically garbage collected.

For further information on events, see Chapter 4, "Events."

NOTE *Soft references can also be used by any NeWS system process that tracks resources within the system. An example of such a process is a window manager, which tracks windows. When all other references to a window are removed, the window manager can respond to the consequent system-generated obsolescence event by removing its own soft reference. This arrangement prevents a useless window from continuing to exist due to its link with the window manager.*

Hard References

A counted reference that is not soft is known as a *hard* reference. Thus, the total number of counted references to an object is the sum of the number of soft and hard references. Just as a hard reference can be made soft with the **soften** operator, a soft reference can be made hard with the **harden** operator (see Section 8.2, "Memory Management Operators").

Reference Tallies

The server keeps the following two tallies for each counted object:

- The total number of counted references to the object
- The number of soft references to the object, which is a subset of the total number of counted references

8.2. Memory Management Operators

This section describes the three memory management operators that the server provides: **soften**, **harden**, and **soft**.

Softening a Reference

The following operator can be used to convert a hard reference to a soft reference:

any **soften** any

This operator takes a single argument and returns it unchanged, except that if the argument is a reference to an object, it is returned as a soft reference. If the operator is used to soften the last existing hard reference to an object, the object becomes obsolete and an obsolescence event is generated by the system.

Server objects are frequently pointed to by many references; thus, when using the **harden**, **soft**, and **soften** operators, the programmer is responsible for specifying the correct reference to be operated upon. The following example demonstrates how to specify a particular reference to an object:

```
/cv framebuffer newcanvas def       % This creates a single hard
                                    % reference in the process'
                                    % dictionary.

cv setcanvas                        % This creates a second hard
                                    % reference, extending from
                                    % the graphicsstate to the
                                    % current canvas.

/cv cv soften def                   % This softens the cv reference.

currentcanvas soften setcanvas      % This softens the graphicsstate
                                    % reference.
```

Hardening a Reference

The following operator can be used to convert a soft reference to a hard reference:

any **harden** any

This operator takes a single argument and returns it unchanged, except that if the argument is a soft reference to an object, a hard reference to the same object is returned.

Caution should be exercised when using this operator; results may not be as expected depending on the state of the target object. For example, suppose the target is an obsolete canvas on which most obsolescence handling has been

performed. If some of the soft references that have been removed belonged to system processes such as the window manager, hardening a remaining soft reference will keep the canvas in existence, but it will not be tracked in the system as it had been.

Determining a Reference's Type

You can use the following operator to determine if a particular reference is hard or soft:

any **soft** boolean

This operator takes a single argument and returns **true** if the argument is a soft reference to the object *any*, **false** otherwise.

8.3. Memory Management Debugging Operators

This section describes the operators that the server provides for memory management debugging.

With the exception of the operator **vmstatus**, the debugging operators described in this section are contained in a system dictionary named **debugdict**. Therefore, before typing any of these debugging operators in the `psh` connection, you must type debugdict begin to place **debugdict** on the dictionary stack. When you type end, the dictionary is closed. If you attempt to use the debugging operators while the dictionary is closed, the system signals `undefined` errors. Thus, you must use the debugging operators as follows:

```
debugdict begin
 debugging operators
 .
 .
end
```

The debugging operators are described below.

Counting the Number of Server Objects

You can use the following operator to count the number of objects currently in the server:

file **objectdump** –

Writes to the specified *file* a formatted summary of the number of objects that the server has created. Note that *file* must be open for writing; otherwise, an `invalidaccess` error is signaled.

In the output, objects are classified according to the following *families*:

- *interpreter data*

  Objects allocated by the interpreter for the execution of POSTSCRIPT language code.

- *core types*

  Objects allocated for NeWS language processes; any of these objects can appear on the operand stack of a process.

- *shapes classes*

  Objects allocated by the underlying graphics library to perform rendering.

- *miscellaneous types*

  Objects allocated for system management, such as the overhead incurred processing fonts and the memory allocated to support the unused font cache (see Section 8.6, "The Unused Font Cache," for a discussion about the font cache).

- *other*

  Objects for which accounting is not performed, I/O buffer space, and memory allocator overhead.

The output written to the specified file has the following form:

```
family_name family:
    nnnnn bytes for mm object_type objects
```

The **objectdump** operator is demonstrated by the following psh example:

```
debugdict begin                      % Use the debugging dictionary.
currentfile objectdump               % This directs output to the
                                     % psh connection.

(/tmp/objects1) (w) file objectdump  % This directs output to the
                                     % specified file.
/new MyClass send
(/tmp/objects2) (w) file objectdump  % A second file is specified; the
                                     % two files can now be compared
                                     % to indicate the number of
                                     % objects created due to the
                                     % creation of an instance
                                     % of MyClass.

end                                  % End use of the debugging
                                     % dictionary.
```

Returning an Object's Reference Count

The following operator returns the number of references to an object:

object **refcnt** fixed fixed

Returns two numbers onto the process stack: the first is the total reference count for *object*, and the second is the soft reference count for *object*. (Note that the

counts indicate the status of the object after the operator has cleaned up the reference to the object that was given to it on the stack.)

Printing Information on All Current References

You can use the following operator to print information on all the current references in the server:

object **reffinder** –
object boolean **reffinder** –

Prints to standard output information on all current references to the specified *object*. The optional *boolean* argument can be **true** (which requests information only about references that are not soft) or **false** (which requests information about all references to the object). If the specified object is not a counted type, a message is printed and **reffinder** returns.

The **reffinder** operator causes memory to be allocated for a hash table, which holds traceback information about the system. All allocated memory is freed when the operator returns. If memory cannot be allocated, a message is printed, all memory currently allocated due to the operator is freed, and the operator returns.

The **refcnt** and **reffinder** operators can be used together, where **refcnt** determines how many references to an object continue to exist, and **reffinder** prints information on those references. Note that if a call to **refcnt** indicates that all but one of an object's remaining references are soft, the problem you are debugging is likely to have been caused by the remaining non-soft reference. In such a case, **reffinder** should be executed with the *boolean* argument specified as **true** so that only information on the non-soft reference is printed.

Use of **reffinder** may indicate a discrepancy between the number of references registered in the object's tallies and the number that actually exist in the system. If this situation occurs, messages are printed to indicate the discrepancy. There are at least two possible reasons why the discrepancy might occur:

- A *cycle* exists; that is, an object in the system contains a reference to itself and there are no external references to the object. Note that the **reffinder** operator cannot find cycles.

  The effects of a cycle can be illustrated by the following example. Suppose the **reffinder** operator is used to search for references to a canvas, and that a reference to the canvas is held by a dictionary. Also suppose that no external references to the dictionary exist, but that the dictionary holds a reference to itself (for example, if a key in the dictionary is an array with an element that references the dictionary). In this case, **reffinder** cannot find the dictionary since there are no external references to it. Since it cannot find the dictionary, it cannot find the dictionary's reference to the canvas.

- A reference counting bug exists in the server. Although possible, the likelihood of this happening is small. User code should be examined thoroughly for cycles and errors in cleanup processing.

If such a discrepancy is reported, proceed as follows:

- If all existing references to the object are soft, check the obsolescence processing to be sure it is being invoked correctly and that, once invoked, it is executing correctly.
- Check code (particularly POSTSCRIPT language code) for reference counting bugs and cycles.

Inspecting Memory Usage

You can use the following operator to retrieve information about memory usage:

– **vmstatus** num num num

This operator returns three numbers, which indicate the amount of available memory, the amount of memory used, and the system break value.

NOTE *The information returned by this NeWS operator differs from that returned by the standard POSTSCRIPT language* **vmstatus** *operator.*

The **vmstatus** *operator is provided as a standard NeWS operator and is not part of* **debugdict**, *as are the other operators described in this section.*

8.4. Memory Management Debugging Tools

In addition to the NeWS operators discussed in the previous section, the following three shell commands are available for debugging memory problems:

- `objectdiff`

  `objectdiff` performs a `diff(1)` on two files that contain output from the NeWS operator `objectdump`; the `objectdiff` tool produces a formatted summary of the differences in the number and/or size of the objects of each type.

  If the files were not produced by using `objectdump` on the same server during the same run, the output may be meaningless. If the two files were produced by using `objectdump` on two different releases of the server, a change in accounting could cause synchronization problems.

  Refer to the `objectdiff` manual page in the *X11/NeWS Server Guide* for details and examples.

- `objectwatcher`

  `objectwatcher` is a Unix Bourne shell script that prints a formatted summary of data objects allocated and deallocated in the server since the command was last run. This tool uses the `objectdump` operator and the `objectdiff` command.

  The first time `objectwatcher` is run, there should be no output because there should be no file that contains information from a previous run. If a file exists from a run on a previous server, the output from `objectwatcher` is meaningless.

  Refer to the `objectwatcher` manual page in the *X11/NeWS Server Guide* for details and examples.

- `psps`

  `psps` is a `psh` script that prints information about every lightweight process

in the server.

Refer to the `psps` manual page in the *X11/NeWS Server Guide* for details about the information provided.

8.5. Hints for Debugging Memory Leaks

A *memory leak* occurs when memory is allocated but never freed. This section describes how to identify a memory leak and how to gather the data needed to fix the leak.

The server/client model influences resource management in the system: resources are allocated as needed to support clients. Therefore, the size of the server fluctuates during use. Resources allocated to process a client's requests do not necessarily disappear when that client exits; they may be re-used by subsequent server processing. The specific fluctuation in server size depends on the number of processes run and the type of work performed by each one.

Note that most of the available debugging operators and tools should be used from a physical terminal that is separate from the server's display, as opposed to a window-based terminal such as `psterm`. This separation of processing is important because everything invoked on the X11/NeWS system changes the state of its memory (for example, when you run a debugging tool, the server must allocate resources to gather the requested data and print the appropriate text).

Identifying a Memory Leak

In order to identify a memory leak, you must design a very specific test case. Simply noticing that the size of the server has grown after running multiple applications does not prove that a problem exists. The test case must be confined to specific actions within a specific environment. For example, you could bring up an application and quit it, delete mail messages, resize an image, or run a POSTSCRIPT program.

The actions in your test case must be repeated. The first time any action is performed, some resources that are allocated to support the operation will probably not be destroyed when the operation completes; thus, you must repeat the action to distinguish this normal behavior from a memory leak. This phenomenon occurs for two basic reasons. First, much of the server's processing is not synchronous; that is, the server could be on a subsequent pass through some processing when objects from previous operations are destroyed. Second, many objects in the system are cached for performance reasons; their allocation is seen the first time an operation is performed, but their deallocation is seen later or not at all. Therefore, information obtained from invoking some processing once is not useful.

To identify a leak, you should first use the **vmstatus** operator because it provides general information about memory allocation/deallocation. You should run this operator before and after the operation in question. Using the example of starting and quitting an appliation, **vmstatus** should be run before the application is opened, after it is open, and again after it is quit. This sequence of actions will indicate whether resources allocated for the creation are destroyed when the application dies. (You should repeat this sequence of actions because the first time it is done, much of the memory that is allocated is not freed.)

Gathering Data

If you have identified a memory leak, you should consider whether you are running code (such as an application or POSTSCRIPT code) that is not provided with the product or whether you are running the product as it was shipped. If new code is being run on the server, that code is the most suspect.

The memory management scheme of reference counting is exposed upwards in the system. You must understand the design and implementation of reference counting when you write new POSTSCRIPT language code and applications because operations in your code can indirectly cause memory allocation/deallocation.

In your POSTSCRIPT language code, for example, you could easily leave objects on the stacks, create a cyle, or handle obsolescence and/or destruction processing improperly. All these mistakes affect reference-counted objects; you should look for these problems when debugging.

In your C code, you might request server resources and never deallocate them. In this case, each time a particular application action occurs, server resources are again allocated but not deallocated, resulting in continual growth of the server. When the application is killed, these extra server resources may or may not be cleaned up, depending on the resources and the processing involved.

Once a small test case has been identified, you should run `objectwatcher` between multiple invocations of the test case to determine the specific object types being leaked. The type of object is important. Processes and rooted canvases, in particular, reference many other objects. If a process or rooted canvas is leaking, everything that it references is also leaking.

If a process is leaking, you should run the `psps` command to determine which process is the problem. Then you can use the `getprocesses` operator to obtain a reference to the process. If a process is being leaked, you should carefully check obsolescence and destruction processing for bugs.

If a canvas is staying on the screen after some processing has exited, you can use the **canvasesunderpoint** operator to obtain a reference to the canvas. Then you can use the **reffinder** operator to gather information about the objects still holding references to the canvas.

If other object types are being leaked, you should use **reffinder** to determine what objects are holding the references. If the references are on the stack, you should check your code for proper clean up.

When debugging X11 applications, you can use the `xscope` tool, if it is available, to determine what X requests are being generated during the processing that is causing the leak.

Filing Bug Reports

If your data indicates the presence of a leak in the server, you should file a bug report that contains the following information:

- A description of a specific, reproducible test case.
- The data from **vmstatus**, `objectwatcher`, and other tools; be specific about how much memory is leaking and what object types are involved.

- The specific data about the environment, including the server version, the machine configuration, what is running on the system besides the server, and whether you are using customized POSTSCRIPT code.

8.6. The Unused Font Cache

The memory requirements for font representation may be high, particularly when an application uses multiple fonts or employs a wide range of font sizes. It is important, therefore, that no font ever occupy memory unnecessarily.

However, it is often inappropriate to remove a font from memory when its reference count becomes zero: the font may need to be used again, and the performance cost of reloading fonts is high. Therefore, to prevent the unnecessary reloading of fonts, the server provides an *unused font cache.* When a font's reference count becomes zero, it is not destroyed; instead, it is placed in the unused font cache. The font thus continues to exist in memory. When the font is subsequently referenced, it is removed from the cache and is again available for use.

The size of the cache is limited at all times; the limit can be determined by the user (see the following subsection for details). If the cache becomes full and a new font needs to be added, the earliest cached font is removed and destroyed; its memory is thus freed. Fonts continue to be removed from the cache and deleted from memory until sufficient room for the new font has been created. (Note that fonts may be of different sizes and may thus maintain different memory requirements; therefore, no precise figure exists for the number of fonts that may be cached at one time.)

The size limit of the unused font cache determines the balance between memory consumption (caused by maintaining fonts in memory) and performance degradation (caused by reloading fonts). When the size limit is high, fonts tend to be maintained in memory; when the limit is low, fonts tend to be destroyed.

Setting and Inspecting the Size of the Cache

Memory is not allocated for the unused font cache. The size of the cache is the amount of space in the system consumed by unused fonts; it is set to a default value during system initialization. The following operators can be used to query and set the size of the cache:

– **currentfontmem** num

This operator returns the size of the font memory cache in units of kilobytes. This number represents the amount of memory that is used to store unused fonts in the system.

num **setfontmem** –

This operator sets the size of the font memory number. The *num* argument specifies the size of the cache in units of kilobytes. This number is the amount of memory that is used to store unused fonts in the system.

If a font that is bigger than the current size limit of the cache is itself cached, the cache is automatically expanded by the size of the new font. After this has occurred, the size of the cache can only be decreased by use of the **setfontmem** operator.

For most font uses, the default cache size is sufficient. However, if you run applications that use multiple fonts, and some of the fonts are large, an expanded cache may be required to avoid the appearance of performance degradation.

Flushing the cache

To flush the cache, execute **setfontmem** with *size* set to 0. This frees the memory of all unused fonts. If the system is run with the cache size set to zero, the memory of each font is freed whenever its reference counts go to zero. Running the system with a cache size of zero is not recommended, due to the performance penalties associated with loading fonts.

Applications

Applications cannot ascertain current system memory; thus, they should never attempt to modify the cache size. Setting an improperly high cache size may consume all available memory and cause the server to crash. The operators for setting and inspecting the size of the font cache are intended for users only; users can adjust the size of the font cache to be appropriate for their specific environment.

9

NeWS Type Extensions

NeWS extends the POSTSCRIPT language with a number of new types. These new types are necessary because NeWS programs run in a dynamic, interactive environment whereas traditional POSTSCRIPT programs run inside a printer. The type extensions allow NeWS to support multiple imaging surfaces, user input, multiple processes, and the other requirements of a window system.

In addition to the type extensions, NeWS defines a number of operator extensions to support the new types. The operator extensions are described in the next chapter.

Some of the NeWS type extensions are opaque and can only be used with operators that have been created or extended to handle them. Other types behave just like dictionaries, and all dictionary access operators can be used on them. This chapter describes all the NeWS type extensions.

9.1. NeWS Objects as Dictionaries

Some NeWS type extensions have pieces of internal state that are accessible to the NeWS programmer. Objects of these types behave almost exactly like standard POSTSCRIPT language dictionary objects. All of the standard dictionary manipulation operators (such as **begin**, **def**, **get**, and **put**) work on these new types. However, the internal representation of these objects is completely different from standard dictionaries, and storing or retrieving values from these new types may involve side-effects. Objects of these new types are known as *magic dictionary* objects.

Although magic dictionaries are extremely similar to standard PostScript language dictionaries, several important differences exist:

- Magic dictionary objects are not created with the **dict** operator as are standard dictionaries. Instead, magic dictionary objects are created with special operators. NeWS provides one creation operator for each magic dictionary type. For example, the **newcanvas** operator creates a new canvas, and the **createevent** operator creates a new event.
- Magic dictionary objects contain predefined key-value pairs, which cannot be removed with the **undef** operator. The key in each pair names a piece of internal state of the object, and the value is a POSTSCRIPT language representation of that state. These predefined keys are already present in newly-created magic dictionary objects.

- Some of the predefined key-value pairs are read-only. Attempts to write these key-value pairs (for example, with **put** or **def**) will result in an `invalidaccess` error.

The following examples illustrate the use of operator extensions and standard dictionary operators to manipulate magic dictionary objects. The examples use **canvastype** and **eventtype** objects. These types are explained in greater detail later in this chapter.

```
/MyCanvas framebuffer newcanvas def   % Create a new canvas as a child of
                                      % the framebuffer and store it in
                                      % /MyCanvas in the current
                                      % dictionary.

MyCanvas /Mapped true put             % Set the mapped state of MyCanvas
                                      % to be true. This has the side
                                      % effect of painting the contents
                                      % of the canvas to the screen.

MyCanvas /Color get                   % Retrieve the Color attribute of the
                                      % canvas, a boolean value. This
                                      % attribute is read-only and cannot
                                      % be changed.
```

9.2. List of NeWS Types

This section lists all the types that are accessible to NeWS programmers.

POSTSCRIPT Language Types

NeWS provides the following standard POSTSCRIPT language object types, which are returned by the **type** operator:

Table 9-1 *Standard Object Types in the POSTSCRIPT Language*

arraytype	**marktype**	**realtype**
booleantype	**nametype**	**savetype**
dicttype	**nulltype**	**stringtype**
filetype	**operatortype**	
integertype	**packedarraytype**	

All of the above types, except **packedarraytype**, are described in the *PostScript Language Reference Manual*. The **packedarraytype** is a new POSTSCRIPT type that will be included in a future edition of the *PostScript Language Reference Manual*.

Note that the NeWS language supports the new type of POSTSCRIPT language name known as an immediately evaluated name. This new type of name is represented by the **//name** token. When the scanner encounters a **//name** token, it immediately looks up the name in the current dictionary stack and substitutes the corresponding value for the name. The name's value is substituted, not executed, just as if the **load** operator had been used on the name. If the name is not found, an `undefined` error occurs. The **//name** token is commonly used in

procedures to tightly bind names to their values.

NeWS Type Extensions

The following types are provided by NeWS as extensions to the POSTSCRIPT language:

Table 9-2 *Additional NeWS Object Types*

canvastype	**environmenttype**	**pathtype**
colormapentrytype	**eventtype**	**processtype**
colormaptype	**fonttype**	**visualtype**
colortype	**graphicsstatetype**	
cursortype	**monitortype**	

All of the above types are accessible as POSTSCRIPT language dictionaries except for **colortype**, **graphicsstatetype**, **monitortype**, and **pathtype**.

The **type** operator returns the name of all of the above NeWS type extensions except for **fonttype**; the **type** operator returns **dicttype** for a NeWS font object to be consistent with the POSTSCRIPT language. The NeWS **truetype** operator returns **fonttype** for a NeWS font object.

The sections that follow provide a description of each NeWS type extension. For each type that is accessible as a dictionary, the dictionary keys are described. The types that are not accessible as dictionaries are listed first; the other types are listed in alphabetical order. Because **packedarraytype** is not yet described in the *PostScript Language Reference Manual*, it is described here along with the NeWS type extensions.

9.3. colortype

NeWS *color* objects can have either *red/green/blue* or *hue/saturation/brightness* values. Color objects with red/green/blue components are created with the **rgbcolor** operator. Color objects with hue/saturation/brightness components are created with the **hsbcolor** operator. The color objects can be compared and can be used as a source of paint for the rendering primitives. Color objects cannot be accessed as dictionaries.

NOTE *NeWS provides a dictionary of named colors; see Chapter 11, Extensibility through NeWS Procedure Files for information.*

9.4. graphicsstatetype

Graphics state objects preserve entire graphics states, as defined by the POSTSCRIPT language, in a permanent form. Their only use is to save the graphics state of a process for future re-use by that (or another) process. They are retrieved and set with the **currentstate** and **setstate** operators. They cannot be accessed as dictionaries.

9.5. monitortype

Monitor objects can be accessed by only one process at a time; they are used for synchronization. A monitor object can be locked or unlocked. Processes can use monitors to implement mutual exclusion (for example, to prevent conflicts in updating shared data structures). Monitors are created with the **createmonitor** operator. Monitors cannot be accessed as dictionaries.

9.6. packedarraytype

The NeWS **packedarraytype** is equivalent to the new POSTSCRIPT **packedarraytype**; **packedarraytype** is documented here because it is not yet included in the *PostScript Language Reference Manual.*

A **packedarraytype** object is a more compact representation of an array than an ordinary **arraytype** object. *Packed arrays* save space; they should be used whenever possible.

In many ways, packed array objects are similar to ordinary array objects. Packed arrays can be executed. Elements or subarrays can be extracted from a packed array with the standard **get** and **getinterval** operators; a subarray of a packed array is itself a packed array. Packed arrays can be enumerated with the **forall** operator.

However, some differences do exist between packed array objects and ordinary array objects. Packed arrays are always read-only; the **put** operator cannot be used to store into a packed array. Accessing arbitrary elements of a packed array can be slow, but accessing the elements sequentially takes about the same amount of time as it does for an ordinary array.

The **setpacking** operator can be used to set a process' array-packing mode to true; when true, the server automatically creates packed arrays for any executable array that it reads for that process. When the symbol "{" is encountered, the server accumulates all tokens until the associated "}" and then creates a packed array instead of an ordinary array. The array-packing mode defaults to false. A child process inherits its parent's array-packing mode.

A packed array can also be created with the **packedarray** operator. This operator takes as arguments the objects that are to be included in the packed array.

9.7. pathtype

Path objects represent paths, as defined by the POSTSCRIPT language, in a permanent form. Their only use is to save the current path of a process for future re-use by that (or another) process. They are retrieved and set with the **currentpath** and **setpath** operators. They cannot be accessed as dictionaries.

9.8. canvastype

All NeWS *canvas* objects are of type **canvastype**. Each canvas is a surface on which objects such as text or graphic images can be drawn. A canvas' boundary is represented by a POSTSCRIPT language path and can be any arbitrary shape. When mapped to the screen, canvases can overlap. When this occurs, the hidden portion of a canvas can be stored offscreen and redisplayed when the canvas is re-exposed.

Canvases exist in a hierarchy. The background of the screen is the root of the hierarchy and is thus known as the *root canvas.* A canvas can have any number of children, each of which can exist at any coordinates; however, each child is visually clipped by the bounds of its parent and thus becomes invisible when located outside those bounds.

Canvases are created with the **newcanvas** operator. They can be accessed as dictionaries.

Canvases are described in detail in Chapter 2, *Canvases*. This section describes the keys in the canvas dictionary.

A **canvastype** dictionary contains the following keys:

TopCanvas
BottomCanvas
CanvasAbove
CanvasBelow
TopChild
Parent
Transparent
Mapped
Retained
SaveBehind
Color
EventsConsumed
Interests
Cursor
Colormap
Visual
VisualList
VisibilityInterest
SubstructureRedirect
OverrideRedirect
BorderWidth
XID
SharedFile
RowBytes
Grabbed
GrabToken

The value of each key is described below.

TopCanvas canvas (*read-only*)
The canvas' top sibling (the **TopChild** of the parent canvas), or the canvas itself if it has no siblings.

BottomCanvas canvas (*read-only*)
The canvas' bottom sibling (the bottom child of the parent canvas), or the canvas itself if it has no siblings.

CanvasAbove canvas *or* null
The sibling canvas immediately above this canvas, or null if no such canvas exists. You can change a canvas' position in the hierarchy by setting the value of this key to be any of the canvas' siblings. When you set the value of a canvas' **CanvasAbove** key, the canvas is inserted into the hierarchy directly below the specified sibling. Note that the **CanvasAbove** and **CanvasBelow** keys of the affected siblings will change to reflect the new hierarchy.

CanvasBelow canvas *or* null
The sibling canvas immediately below this canvas, or null if no such canvas exists. You can change a canvas' position in the hierarchy by setting the value of this key to be any of the canvas' siblings. When you set the value of a canvas' **CanvasBelow** key, the canvas is inserted into the hierarchy directly above the specified sibling. Note that the **CanvasAbove** and **CanvasBelow** keys of the affected siblings will change to reflect the new hierarchy.

TopChild canvas *or* null (*read-only*)
The top child of this canvas, or null if no such canvas exists.

Parent canvas *or* null
The parent of this canvas, or null if the canvas has no parent. Null is associated with canvases that result from **createdevice**, **readcanvas**, and **buildimage**. Setting a canvas' **Parent** key manipulates the canvas hierarchy; the canvas becomes the top child of the canvas specified in this key. Canvases created with **readcanvas** and **buildimage** cannot be inserted into the canvas hierarchy; setting the **Parent** key of such a canvas is ignored.

Transparent boolean
True if the canvas is transparent, false if it is opaque. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. An opaque canvas can be damaged; a transparent canvas cannot. A transparent canvas never has a retained image; instead it shares its parent's retained image. Anything painted on a transparent canvas is actually painted on the first opaque canvas beneath it (often, its parent).

Mapped boolean
True if the canvas is mapped, false if it is unmapped. When a canvas is mapped, it becomes visible on the screen that its parent is on, provided that all of its ancestors are mapped and that it is not obscured by overlapping canvases. Note that canvases created with **readcanvas** and **buildimage** cannot be mapped to the screen. When a nonretained canvas is mapped, the region that becomes visible is considered to be damaged.

Retained boolean
True if the canvas is retained, false if it is not. NeWS keeps an offscreen copy of the invisible parts of a retained canvas. If a retained canvas is mapped and is overlapped by some other canvas, the hidden parts of the canvas will be saved. If a canvas is retained when it is not mapped, a copy of the entire canvas is saved.

A retained canvas usually performs much better with most window management operations, like moving and mapping canvases. But the retained image does consume storage. For color displays, the cost of retaining canvases is often prohibitive.

If the server runs low on memory, the retained portions of canvases may be reclaimed. When this happens, querying the **Retained** field of such a canvas returns false. In addition, damage may be reported on this canvas. Therefore, programs should be prepared to handle damage on any canvas, including retained

ones.

The **Retained** field is meaningless for a transparent canvas. When queried, it returns the **Retained** value of its nearest opaque ancestor; in this case, the value cannot be changed.

SaveBehind boolean

SaveBehind is a hint to the window system that when the canvas is made visible on the screen, the canvas won't be up very long and the canvases below it won't be very active. If the value of a canvas' **SaveBehind** key is true, NeWS usually saves the values of the pixels underneath the canvas when the canvas is mapped to the screen. NeWS then restores the original pixel values back to the screen when the canvas is unmapped, and none of the canvases are damaged. This is a performance hint only; it does not affect the semantics of any other operations. It is usually employed with pop-up canvases to reduce the cost of damage repair when they are unmapped.

Color boolean *(read-only)*

True if and only if this canvas can support more colors than just black-and-white or greyscale.

EventsConsumed name

This key determines the event consumption behavior of the canvas. Its value is one of the following names:

/AllEvents

All events that are tested against this canvas' post-child interests are consumed; they are not tested against the post-child interest lists of this canvas' ancestors.

/MatchedEvents

Events that match a post-child interest of this canvas are consumed, but non-matching events may still pass to this canvas' ancestors for further testing against post-child interests.

/NoEvents

No events are consumed by this canvas; all events may pass to the canvas' ancestors during testing against post-child interests.

Interests array *(read-only)*

The interest lists for the canvas, represented as an array of events. The array is a concatenation of the canvas' pre-child and post-child interest lists, with the pre-child interest list first. Within each list, the interests are ordered according to their priority, with highest priority first. Among interests with the same priority, exclusive interests are listed first.

Cursor cursor *or* null
The cursor associated with this canvas, or null if a cursor has not been specified for this canvas.

Colormap colormap
The colormap that is associated with this canvas (see **colormaptype**).

Visual visual (*read-only*)
The visual that is associated with this canvas (see **visualtype**).

VisualList array (*read-only*)
An array that contains all possible visuals for the canvas (see **visualtype**).

VisibilityInterest boolean (*read-only*)
For X windows, true if the canvas is interested in visibility changes. (This key is useful only for canvases created by X11.)

SubstructureRedirect (*read-only*)
For X clients, SubstructureRedirect is a reference to the process that selected substructure redirect on the canvas. This is usually found only on framebuffers. (This key is useful only for canvases created by X11.)

OverrideRedirect boolean (*read-only*)
True if an X11 client has selected the **OverrideRedirect** window attribute for this canvas. (This key is useful only for canvases created by X11.)

BorderWidth null *or* integer (*read-only*)
The X11 border width. If this value is an integer, the canvas has a window border with the specified width. A non-null **BorderWidth** can be changed with the **reshapecanvas** operator. If the value is null, the canvas has no border and none can be set. (This key is useful only for canvases created by X11.)

XID number (*read-only*)
The X11 resource ID of the canvas. If this number is zero, the canvas is not in the X11 resource database. (This key is useful only for canvases created by X11.)

SharedFile string
Associates the canvas object with a file; the contents of the file become the contents of the canvas. The canvas must be an unrooted canvas. If the canvas does have a parent, a `typecheck` error is returned. The *string* must contain the name of a file in the server's name space. If the file is inaccessible or does not have read-write access permission, an `invalidfileaccess` error is returned. If the canvas currently has a non-null **SharedFile** value, changing the value to null disassociates the file and the canvas; changing the value to a filename disassociates the current file and associates the canvas with the newly specified file. The file is assumed to contain image data stored a line at a time in increasing y order, the number of bytes per scanline being that specified by **RowBytes**. Note that the contents of a canvas are lost the first time its **SharedFile** key is set.

The ability to map a canvas to a file is operating system dependent and may not be present in the server.

The file should be accessed by the client directly using mmap(2). The server will process the shared file in native byte order. The client is responsible for synchronizing accesses to the shared file. This facility is intended for use by clients running locally with the server. If the client and server do not reside on the same machine, canvas data consistency is not guaranteed by the server.

RowBytes number (*read-only*)
The scanline padding requirements for a canvas. This represents the dimensioned width plus any padding added by the server.

Grabbed boolean
Unless you are using a GX graphics accelerator, neither this key nor the **GrabToken** key (see below) has an effect on the canvas.

If you are running X11/NeWS with a GX graphics accelerator (FRAMEBUFFER=/dev/cgsix0), this key controls NeWS access to the graphics hardware. When used in conjunction with a C language interface to the hardware, the key mediates the control over the bits inside a given canvas. To demonstrate how the **Grabbed** key is used, the following code creates a new canvas:

```
/can framebuffer newcanvas def
```

The following example shows the three possible uses of the **Grabbed** key:

```
can /Grabbed true put          % Make can a grabbed window.
can /Grabbed false put         % Release the grab on can.
can /Grabbed get               % Returns the value of
                               % the Grabbed key.
```

When a GX graphics accelerator is present and a client sets a canvas' **Grabbed** key to true, the cgsix segment driver assigns an integer to the canvas' **GrabToken** key. The client can then communicate with the cgsix segment driver using this **GrabToken** to identify which canvas' clip area to use when rendering directly to the framebuffer.

GrabToken int (*read-only*)
The grab token for the canvas. This key's value is zero when the canvas is not grabbed and is a non-zero integer when it is grabbed. The key is demonstrated by the following example:

```
can /GrabToken get             % Returns 0 if not grabbed.
```

9.9. colormaptype

A *colormap* is a color lookup table that determines which color is displayed for a specified pixel value. Each entry in the colormap table contains a red, a blue, and a green value; these values can be used to specify the color-mix of a given pixel. Each entry also contains an integer that is a index for the entry.

A colormap can be created with the **createcolormap** operator (see Chapter 10, *NeWS Operator Extensions*). Colormaps can be accessed as dictionaries.

A **colormaptype** dictionary contains the following keys:

Entries
Free
Installed
Visual

The value of each key is described below.

Entries array (*read-only*)
An array of the colormapentries used by this colormap. The mininum number of elements in the array is 0; the maximum number of elements is given by the **Size** key of the colormap's visual.

Free number (*read-only*)
The number of free entries in the colormap.

Installed boolean
True if the colormap is installed as a hardware map; otherwise false.

Visual object (*read-only*)
An object that is the visual for this colormap. Note that a canvas and its colormap must have the same visual. The colormap's visual is specified as an argument to the **createcolormap** operator.

9.10. colormapentrytype

A *colormapentry* is usually a single entry in a colormap; however, it may also be specified as a group of several entries (or *slots*). In such cases, a bitmask can be used to manipulate the indices of the entries and thereby derive the required color. Colormapentry objects can be accessed as dictionaries.

A colormapentry is created with the **createcolorsegment** operator. The colors of a colormapentry are accessed with **putcolor** and **getcolor**.

A **colormapentrytype** dictionary contains the following keys:

Colormap
Mask
Slot

The value of each key is described below.

Colormap object (*read-only*)
The colormap to which the entry belongs.

Mask int (*read-only*)
A mask of bits that can be used on a multiple entry to manipulate its indices. If the entry is not a multiple entry, the value of **Mask** is 0.

Slot int (*read-only*)
An integer that is the index position of the slot in the entry. If the entry has only one slot, the value of **Slot** is 0.

9.11. cursortype

Cursor objects are composed of a *cursor image* and a *mask image.* These two images are superimposed to create the complete cursor. (See Table 11-1 for a description of standard NeWS cursors.)

Mask and cursor images each have three attributes: a font, a character within the font, and a color. The cursor image and mask image are superimposed by aligning the origins of their respective characters. This point is also the cursor *hot spot* (the pixel coordinate to which the cursor points).

You can think of the mask image as the background and the cursor image as the foreground. The mask image defines the shape and color of the background on which the cursor image is painted. The mask image is like a stencil that the cursor image is passed through; any parts of the cursor image that fall outside of the mask will not be painted. The portion of the complete cursor painted by the cursor image appears in the cursor image color. The remainder of the complete cursor appears in the mask image color. The complete cursor has a halo effect if a cursor image is superimposed on a larger mask image.

Each canvas in the hierarchy has an associated cursor object specified by its **Cursor** key; the canvas' cursor is displayed when the mouse pointer is over the canvas. When a canvas is created with the **newcanvas** operator, the new canvas inherits the cursor of its parent.

Cursors are created with the **newcursor** operator. A cursor's characters and fonts are determined by the arguments specified to the **newcursor** operator. Cursors can be accessed as dictionaries; a cursor's colors are set with two of the dictionary keys. Cursors are not guaranteed to be displayed with their specified colors because some display devices have color limitations. The mask and image are guaranteed to be painted in contrasting colors, however.

NeWS provides a special font, called **cursorfont**, that includes common cursor shapes and their corresponding masks.

A **cursortype** dictionary contains the following keys:

CursorChar
CursorColor
CursorFont
MaskChar
MaskColor
MaskFont

The value of each key is described below.

CursorChar int (*read-only*)
The integer that corresponds to the character used for the cursor image.

CursorColor object
The color with which the image is painted.

CursorFont object (*read-only*)
The font that is used for the cursor image.

MaskChar int (*read-only*)
The integer that corresponds to the character used for the mask image.

MaskColor object
The color with which the mask image is painted.

MaskFont object (*read-only*)
The font that is used for the mask image.

9.12. environmenttype

Environment objects represent information about the server run-time environment. These objects store information about input devices such as the mouse and keyboard. (The **devicedict** dictionary, for example, contains environment objects for (/dev/kbd), (/dev/mouse) and (/dev/fb).) Every device has its own environment object that can be accessed as a dictionary; information is stored only in the subset of keys that pertain to that particular device. The environment dictionary keys are device dependent.

An environment dictionary is created with the **createdevice** operator.

An **environmenttype** dictionary contains the following keys:

BellDuration
BellPitch
BellPercent
KeyClickPercent
Leds
AutoRepeat
KeyRepeatTime
KeyRepeatThresh
MotionCompression
Threshold
AccelNumerator
AccelDenominator

The value of each key is described below.

BellDuration real *or* integer
The duration of the keyboard bell (in 2^{16} milliseconds).

BellPitch real *or* integer
The pitch of the keyboard bell (in Hz).

BellPercent real *or* integer
The volume of the keyboard bell (0.0=off, 1.0=loudest).

KeyClickPercent real *or* integer
The volume of the keyboard key click (0.0=off, 1.0=loudest).

Leds integer
The status of the keyboard LEDs (a bit mask that determines whether the LEDs are on or off).

AutoRepeat boolean
The status of keyboard auto-repeat (true=on, false=off).

KeyRepeatTime real *or* integer
The keyboard repeat key cycle time (in 2^{16} milliseconds). Determines the speed at which a key will repeat.

KeyRepeatThresh real *or* integer
The keyboard repeat key threshold (in 2^{16} milliseconds). Specifies the amount of time a key must be pressed before it begins to repeat.

MotionCompression boolean
The status of pointer motion compression (true=motion compression on, false=motion compression off). If true and the server falls behind in processing motion events, multiple events may be collapsed into one.

Threshold real
The pointer acceleration threshold. Specifies how fast the pointer must be moved (the threshold number of pixels moved at once) before pointer acceleration takes place.

AccelNumerator real *or* integer
Specifies the numerator for the pointer acceleration multiplier. When acceleration takes place, the pointer speed will be multipied by **AccelNumerator/AccelDenominator**.

AccelDenominator real *or* integer
Specifies the denominator for the pointer acceleration multiplier. (See **AccelNumerator** above.)

9.13. eventtype

Events are NeWS objects, generated by the system and by NeWS processes, that are used for handling input and interprocess communication. The system generates input events to report user actions such as mouse motion and key presses. The server receives information from the input devices, translates the information into NeWS events, and distributes the events to the processes that are interested in them. In addition to input events, the server also generates events that tell processes when a canvas is damaged, when an object becomes obsolete, and when a process dies while it is still referenced. NeWS lightweight processes can also generate events and submit them for distribution.

Event objects are created using the **createevent** operator. System-generated events are created automatically. Events can be accessed as dictionaries.

Events are described in detail in Chapter 4, *Events*. This section describes the keys in the event dictionary.

An **eventtype** dictionary contains the following keys:

Action
Canvas
ClientData
Exclusivity
Interest
IsInterest
IsPreChild
IsQueued
KeyState
Name
Priority
Process
Serial
Synchronous
TimeStamp
TimeStampMS
XLocation
YLocation
Coordinates

The value of each key is described below.

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

Canvas null, canvas, dict, *or* array
In an interest, the **Canvas** key indicates the canvas whose interest list the interest is on (or null if the interest is on the pre-child interest list of the root canvas). The **Canvas** key of an interest may contain an array or dictionary; in this case, the interest is placed on the interest list of each specified canvas. When an event is expressed as an interest, this key becomes read-only.

In an event that is to be distributed, the **Canvas** key determines which canvas interest lists are searched for potential matches. If a single canvas is specified, the event is tested against that canvas' interests and the interests of that canvas' ancestors (according to the rules given in Chapter 4, *Events*). If null is specified, the event is tested against the interests of the canvas directly under the event's location (as determined by the canvas **Coordinates** key) and the interests of that canvas' ancestors. If an array or dictionary of canvases is specified, each canvas and its ancestors are considered in turn.

ClientData object
In either an interest or an event submitted for distribution, this field may hold additional information relating to the event. The server does not set or use the value of this key.

Exclusivity boolean
If the **Exclusivity** key of an interest is true, an event that matches this interest in distribution is not allowed to match any further interests. This key is meaningful only for interests; when an event is expressed as an interest, this key becomes read-only.

Interest event *(read-only)*
This read-only key is set in an event as it is distributed; its value is the interest that the event matched in order to be delivered to its recipient.

IsInterest boolean *(read-only)*
True if the event is currently on some interest list.

IsPreChild boolean
True if the event is on the pre-child interest list of its canvas(es). This key has no effect until the event is expressed as an interest; when the event is expressed as an interest, this key becomes read-only.

IsQueued boolean (*read-only*)
True if the event has been put in the input queue and has not yet been delivered.

KeyState array (*read-only*)
When keyboard translation is on, this array is empty. When translation is off, this array indicates all the keys that were down *at the time the event was distributed.* The array actually contains the **Name** values from events that had an **Action** of **/DownTransition** and that did not have a subsequent event with the same **Name** and an **Action** of **/UpTransition**. In generating this array, the test is executed before a down-event, and after an up-event, so a down-up pair with no intervening events will not be reflected in the **KeyState** array.

This key is meaningless in an interest.

Name object
An arbitrary POSTSCRIPT language object that usually indicates the kind of event. For example, keystrokes have numeric values associated with the **Name** key, corresponding to the ASCII characters (or the keys) that were pressed. Other events have name values associated with the **Name** key, such as **/Damaged** or **/EnterEvent**.

Priority number
Priority is meaningful only for interests. When an event is expressed as an interest, this key becomes read-only. Distributed events are matched against the interests expressed on a canvas in priority order, highest priority first. Among interests with the same priority, interests with the **Exclusivity** key set to true are considered first; among nonexclusive interests of the same priority, the most recently expressed interest is considered first. The default priority is 0; fractional and negative values are allowed. The priority rarely needs to be changed from its default value.

Process null *or* process
The **Process** key can be set prior to sending an event out for distribution. In a distributed event, the **Process** key restricts distribution of the event to the specified process. Distributed events usually have null in their **Process** fields and are matched against interests without restriction. The **Process** key in an interest is set by the **expressinterest** operator to be the process that will own the interest. When an event is expressed as an interest, this key becomes read-only.

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

Synchronous boolean
This key's boolean value is only significant for interests. When an event matches an interest that has its **Synchronous** key set to **true**, the process that holds the matching interest is given a chance to run before the next event is removed from the global event queue; the process is responsible for unblocking the global event queue with the **unblockinputqueue** operator.

TimeStamp number
This numeric value indicates the time an event occurred. A time value is simply the amount of time that has elapsed since the system started, calculated in units of 2^{16} milliseconds.

The current nominal resolution of a time value is 1 ms and the maximun interval is 71,582 minutes (49.7 days).

Events in the global event queue are distributed in **TimeStamp** order, and no event is delivered before the time in its **TimeStamp** field. Thus, a timer event is simply any event handed to **sendevent** with a **TimeStamp** value in the future. This key is ignored in interests.

TimeStampMS integer
Similar to **TimeStamp**, except stores the time in units of milliseconds and is an integer instead of a real number. Useful for precise integer arithmetic with event timing. Note that **TimeStamp** and **TimeStampMS** are merely different representations of the same value. **TimeStampMS** is preferred because it is more accurate.

XLocation number
System events are labeled with the cursor location at the time they are generated; this location is used to determine which canvas interest lists are tested against the event for potential matches. The location is available to recipients and is given with respect to the current transformation matrix. This key accesses the *x* coordinate of the event's location. This key is ignored in interests.

YLocation number
This key accesses the *y* coordinate of the event's location; see the explanation under **XLocation** above. This key is ignored in interests.

Coordinates [x-location y-location]
This key accesses the event's *x* and *y* locations as an array with two elements. The *x* and *y* coordinates are given with respect to the current transformation matrix.

9.14. fonttype

A NeWS *font* object is accessible as a dictionary. The **type** operator returns **dicttype** for a NeWS **font** object; the **truetype** operator returns **fonttype**.

A NeWS **font** dictionary includes all the standard keys for a POSTSCRIPT language font dictionary; it also contains the following NeWS-defined keys:

FontAscent height
Specifies the extent of the font above the baseline.

FontDescent height
Specifies the extent of the font below the baseline.

FontHeight height
Specifies the total height of the font, which is the sum of the **FontAscent** and **FontDescent** values.

WidthArray array (*read-only*)
An array of number pairs that specify the x and y components of the width of each character. The x component of character c is in **WidthArray**(2*c), and the y component is in **WidthArray**(2*c+1). The width components are given in units of the current coordinate system with respect to the origin of the character's coordinate system.

PrinterMatched boolean (*read-only*)
A boolean that determines whether the font is printermatched. The key's value is initialized to the value of the process' printermatch state. The value of a font's **PrinterMatched** key can be set with the **printermatchfont** operator.

9.15. processtype

The NeWS server maintains a set of simultaneously executing lightweight *processes*. Each process object is an individual thread of control with its own graphics context, dictionary stack, execution stack, and operand stack. These lightweight processes all exist in the same address space; two processes can refer to the same object if they can both locate the object. Typically, each connection to the server obtains a separate thread of execution with its own context. A process can create, or *fork*, new processes to form a process group. Processes communicate with each other using NeWS events.

When NeWS first starts to run, it creates a single process that executes the NeWS startup file. At this time, code may be downloaded into the server and many more lightweight processes may start. The process that runs the startup file is the only process that is not created by some earlier process executing the **fork** operator.

When a process executes the **fork** operator, the newly created process is the *child* of the *parent* process that created it. The child proccess inherits its parent's dictionary stack, operand stack, and graphics state. The parent and child start out in the same process group. However, the **newprocessgroup** operator can be used to remove a process from its process group and put it in its own, new process group. Although a child process starts out with the same name space as its parent, each lightweight process can control the extent to which its name space is shared with other processes by pushing and popping dictionaries to and from its private stack.

A process can kill its child processes, or it can wait for them to die and obtain a return value from them. A process can pause to allow other processes to run. NeWS processes can also temporarily suspend themselves and other processes. A process can examine the state of other processes by opening the process objects

that represent them as dictionaries.

A process dictionary contains two special keys in **systemdict**: **$error** and **errordict**. When accessed, they return the **$error** or **errordict** of the current process. To access the **$error** or **errordict** of a different process, use the corresponding magic fields in that process. Note that the **$error** field will always be private to an individual process, containing information about the last error that process encountered, but the **errordict** can be shared between processes since its reference is copied to a child process during a fork.

A **processtype** dictionary contains the following keys:

BindOverride
DictionaryStack
$error
errordict
ErrorCode
ErrorDetailLevel
Execee
ExecutionStack
Interests
OperandStack
PackedArrays
ProcessName
State
Priority
Stdout
Stderr
SendContexts
SendStack

The value of each key is described below.

BindOverride boolean

A boolean that specifies the process' current bindoverride state. By default, **BindOverride** is false. If set to **true**, operators previously bound in procedures are resolved during procedure execution by using the operator's original name. Therefore, a **true** setting causes a dictionary stack search for the name of each operator object encountered during the course of executing a procedure. Also, operators bound during `cps` processing are transformed back to their original names when sent to the server across a connection from a cps client.

DictionaryStack array *(read-only)*

An array that contains the current dictionary stack of the process. The dictionary on the bottom of the stack (the **systemdict**) is array element 0, and the process' **userdict** is array element 1.

$error dict *or* null
A dictionary that contains information about the last error that the process encountered. The dictionary is filled by the **defaulterroraction** primitive when errors occur. This error dictionary is similar to the POSTSCRIPT language **$error** dictionary, but it has one additional key named **message**; if **ErrorDetailLevel** is greater than zero, **message** contains a string that describes the context of the error. If the **defaulterroraction** primitive has not been executed, the value of **$error** will be null.

errordict dict
The **errordict** that is used to resolve the process' errors. This **errordict** is copied to a forked process by the **fork** operator. The initial value of this field is a copy of the NeWS listener's **errordict**, which by default maps each error to the **defaulterroraction** operator.

ErrorCode name
A name that specifies the current errorcode of the process. This key's value is one of the following names:

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

Most of the error codes are standard POSTSCRIPT language error codes. However, the following five are NeWS-specific:

- **accept** indicates that something went wrong when the server tried to accept a connection from a client process.
- **killprocess** indicates that the process has been killed, usually by the **killprocess** operator.
- **none** indicates no error.
- **timeout** indicates that the process has exceeded its time quota without pausing. The NeWS **timeout** is different than the POSTSCRIPT language **timeout** because NeWS interprets **timeout** on a per process basis and each process can avoid **timeout** by using the **pause** operator.
- **unimplemented** indicates that the process has executed an operator that is not currently implemented.

ErrorDetailLevel integer
Controls the amount of detail that is included in the default error handler's error report. Setting **ErrorDetailLevel** to 0 (the default) gives a minimum of error reporting. Setting it to 1 records a more descriptive message in the **$error** dictionary, and setting it to 2 records the contents of the dictionary, execution, and operand stacks in the **$error** dictionary. The following line sets the error detail level to 1:

```
currentprocess /ErrorDetailLevel 1 put
```

Execee object *(read-only)*
The object currently being evaluated (i.e., the top of the process' execution stack).

ExecutionStack array *(read-only)*
The full current execution stack of the process, represented as an array that contains pairs of executable arrays and indices. The executable array at the bottom of the stack is element 0 of the array, and the first index is element 1. The indices indicate which element of the associated array is currently being executed.

Interests array *(read-only)*
An array that contains the current interest list of the process.

OperandStack array (*read-only*)
The full current operand stack of the process, represented as an array. The object on the bottom of the operand stack is element 0 of the array.

PackedArrays boolean
A boolean that specifies the process' array packing mode. If the value is true, packed arrays are created. For more information about the process' array packing mode, see the description of the **setpacking** operator in Chapter 10, "NeWS Operator Extensions."

ProcessName string
This key can be used to store an identifying string that gives the process a name. It defaults to (Unnamed process). This value is not used by anything internal to the server but is useful for debugging. (See the manual page for `psps`.)

State array (*read-only*)
A name that specifies the current execution state of the process. The set of possible results is as follows:

- **breakpoint** indicates that the process is suspended, normally for debugging.
- **dead** indicates that the process is completely dead.
- **input_wait** indicates that the process is waiting on an event.
- **IO_wait** indicates that the process is waiting on input/output.
- **mon_wait** indicates that the process is waiting at a monitor.
- **proc_wait** indicates that the process is waiting for another process to exit.
- **runnable** indicates that the process is running.
- **zombie** indicates that the process has exited, but other processes still have references to it.

Priority int
The scheduler priority of the process. The server has an internal priority and will not execute any process whose priority is less than this value. NeWS lightweight processes whose **Priority** falls below this dynamically-changing priority limit are not scheduled to be run by the server — they are "frozen."

NeWS processes should have no need to change their priority. Process priority is only used by the X11/NeWS server to implement "grabs". Normal processes should have a priority of **UserPriority** (0). Processes that cannot block (such as NeWS support processes) should have a priority of **SystemPriority** (100).

Stdout file
The current standard output file of the process.

Stderr file
The current standard error file of the process.

SendContexts array (*read-only*)
An array that contains the current send stack of the process. The dictionary stack on the bottom of the send stack is element 0 of the array.

SendStack array (*read-only*)
Identical to **SendContexts** but in the reverse order, so that it matches the ordering of the other stacks in a process.

9.16. visualtype

A *visual* is an object that describes the permissible color properties for a canvas. Visuals are accessible as dictionaries. Each available visual is system-supplied and its dictionary is read-only. A canvas' visual can be passed as an argument to the **newcanvas** operator; the canvas then has the properties allowed by the specified visual. If no visual is specified, a default visual is used.

To obtain a list of available visuals, examine the **VisualList** key of the root canvas.

Each colormap is also associated with a visual that is specified when the colormap is created; a colormap's visual is stored in its read-only **Visual** key. Note that a colormap and its canvas must have the same visual.

A **visualtype** dictionary contains the following keys:

Size
Class
BitsPerPixel

The value of each key is described below.

Size number (*read-only*)
The maximum number of colormapentries for colormaps associated with this visual (see **colormapentrytype** and **colormaptype**).

Class number (*read-only*)
A number that indicates the color class of the visual. Visuals are divided into six classes, representing six different types of display hardware. The following list describes the classes and their effects on the mapping between pixel value and visible color. The first line of each description gives the number that is the value of the **Class** key, followed by a name (in parentheses) that is commonly used to describe that color class.

0 (StaticGray)
The pixel value indexes a predefined, read-only colormap. For each colormap cell, the red, green, and blue values are the same, producing a gray image.

1 (GrayScale)
The pixel value indexes a colormap that the client can alter, subject to the restriction that the red, green, and blue values of each cell must always be the same, producing a gray image.

2 (StaticColor)
The pixel value indexes a predefined, read-only colormap. The red, green, and blue values for each cell are server-dependent.

3 (PseudoColor)
The pixel value indexes a colormap that the client can alter. The red, green, and blue values of each cell can be selected arbitrarily.

4 (TrueColor)
The pixel value is divided into sub-fields for red, green, and blue. Each sub-field separately indexes the appropriate primary of a predefined, read-only colormap. The red, green, and blue values for each cell are server-dependent and are selected to provide a nearly linear increasing ramp.

5 (DirectColor)
The pixel value is divided into sub-fields for red, green, and blue. Each sub-field separately indexes the appropriate primary of a colormap that the client can alter.

BitsPerPixel number (*read-only*)
The number of bitplanes used by the canvas.

10

NeWS Operator Extensions

acceptconnection

listenfile **acceptconnection** file

Listens on *listenfile* for a request made by a client UNIX process for a connection with the X11/NeWS server. When the request is successfully accepted by the server, the operator returns a file object, *file*, that represents the connection to the client. Information written to *file* is sent to the UNIX client process. Information sent by the UNIX client process to the server can be read from *file*.

The *listenfile* is created by invoking **file** with the special file name `(%socketl`*n*`)`, where *n* is the IP port number used for listening.

See also: **getsocketpeername**

arccos

num **arccos** num

Computes the arc cosine in degrees of *num*.

arcsin

num **arcsin** num

Computes the arc sine in degrees of *num*.

assert

boolean errorname **assert** –

Generates a POSTSCRIPT error of type *errorname* if *boolean* is *false* .

awaitevent

– **awaitevent** event

Removes an event from the head of the current process' local input queue, then places the event on the process' operand stack. If the local input queue does not contain an event, **awaitevent** blocks until an event is placed on the queue: an event is placed on the queue when a distributed event successfully matches an interest expressed by the process.

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

beep

– **beep** –

Generates an audible signal. On most server implementations, this rings the keyboard bell.

blockinputqueue

num *or* null **blockinputqueue** –

Inhibits distribution of events from the global event queue. When the operator is executed, a release time is calculated for the block; the release time is the sum of the current time and the argument to **blockinputqueue**. The argument can be *num* or *null*; *num* is a number in units of 2^{16} milliseconds and *null* represents a system-defined default timeout. When the operator is executed, no event is removed from the global event queue until one of the following has occurred:

- The amount of time specified by the release time has elapsed.
- The **unblockinputqueue** operator is executed.

When nested calls to **blockinputqueue** are made, no event is removed from the global event queue until each of the locks is released. Each lock may be released either when its release time has expired or when a corresponding **unblockinputqueue** operator has been executed once for each call to **blockinputqueue**.

Since an event used as the argument to **sendevent** is inserted in the global event queue, its distribution can be inhibited by **blockinputqueue**. However, an event used as the argument to **redistributeevent** is not inserted in the global event queue; thus, its distribution cannot be inhibited by **blockinputqueue**.

See also: **sendevent, unblockinputqueue**

breakpoint

– **breakpoint** –

Suspends the current process.

buildimage

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

Constructs a canvas object, using the *width*, *height*, *bits/sample*, and *proc* arguments as does the POSTSCRIPT language **image** operator. The parameters represent a sampled image that is a rectangular array of *width* by *height* sample values. Each value consists of *bits/sample* bits of data (1,2,4,8). The data is received as a sequence of characters (that is, 8-bit integers in the range 0 to 255). If *bits/sample* is less than 8, the sample bits are packed left to right within a character (from the high-order bit to the low-order bit). Each row is padded out to a character boundary.

The **buildimage** operator executes *proc* repeatedly to obtain the image data. The specified *proc* must place on the operand stack a string containing any number of additional characters of sample data.

If *proc* is null, **buildimage** constructs the canvas but does not initialize its contents. (This is the recommended way of creating canvases to hold offscreen images.)

The canvas object that **buildimage** creates is retained, has no parent, and is not mapped. The canvas object cannot be mapped: it can be rendered to the screen with the **imagecanvas** or **imagemaskcanvas** operators; it can also be written to a

file with the **writecanvas** operator. The *matrix* argument is used to define the default coordinate system of the canvas.

See also: **imagecanvas, imagemaskcanvas , writecanvas**

canvasesunderpath

– **canvasesunderpath** array

Returns a nested array of canvases that "intersect" the current path, starting with the current canvas. A canvas "intersects" the path if the canvas itself or any of its children fall within the area described by the path. Both opaque and transparent canvases can intersect the path. An opaque canvas can also "consume" the path; that is, prevent any younger siblings that it visually obscures from themselves intersecting the path. A transparent canvas cannot consume the path.

The returned array has the following format:

[*parent* [*child* [..] *child* [..] ..]]

The array is a nested array whose first element is the parent canvas that either intersects the path or has one or more children that themselves intersect the path. The second element is an array whose elements are the children that intersect the path. If a child itself has children that intersect the path, those children appear in a subarray in the position immediately after the child itself.

Note the following examples:

- No canvas intersects the current path:

  []

- The current canvas (A) intersects the current path:

  `[A [] ]`

- The current canvas (A) and one of its children (B) intersect the current path:

  `[A [B [] ] ]`

- The current canvas (A) and two of its children (B and C) intersect the current path:

  `[A [B [] C [] ] ]`

- A canvas (A), its child (B), and grandchild (C) intersect the current path:

  `[A [B [C [] ] ] ]`

- A canvas (A), three children (B, E, and F), and three grandchildren (C, D, and G) intersect the current path:

  `[A [B [C [] D [] ] E [] F [G [] ] ] ]`

canvasesunderpoint

x y *or* null **canvasesunderpoint** array

Returns an array containing the canvas under the given point and all the canvas' ancestors. The ordering is from leaf to root; thus, the canvas under the point is the first canvas in the array, and the root canvas is the last canvas in the array. If *null* is specified instead of *x*, *y*, the operator returns the hierarchy of the canvas that was under the cursor position when the last event was distributed from the global input queue, provided that the event contained meaningful cursor coordinates.

NOTE *This operator does not return canvases that lie geometrically under the given point. The operator describes a canvas' ancestry, returning its parent canvas, its grandparent canvas, and so forth. This can be used to determine how default event distribution takes place from a given canvas.*

See also: **currentcursorlocation**

canvastobottom

canvas **canvastobottom** –

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

See also: **insertcanvasbelow**

canvastotop

canvas **canvastotop** –

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

See also: **insertcanvasabove**

clearsendcontexts

– **clearsendcontexts** –

Removes all history of currently executing send contexts from the current process. This includes classes on the dictionary stack as well as the history of send contexts. This operator essentially removes the process from the influence and context of any current send.

This operator is useful when no return from a **send** is possible, as in a forked process.

See also: **send**

clipcanvas

– **clipcanvas** –

The **clipcanvas** operator is identical to **clip**, except that it sets a clipping path that is an attribute of the current canvas, rather than of the current graphics state. The operator imposes clipping restrictions on all painting operations aimed at the current canvas. This is typically used during damage repair to restrict update operations to the damaged region. If the current path is empty, **clipcanvas** removes the clipping restriction of the current canvas, if such a restriction exists. Note that **clipcanvas** does not intersect the current path with the existing canvas clipping region, as the **clip** operator does.

The clipping boundary set by this operator is not affected by **initgraphics**, **initclip**, **gsave**, **grestore**, or any of the other graphics state modifiers. Graphics operations are clipped to the intersection of the canvas clip, the graphics state clip, and the shape of the canvas.

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

See also: **damagepath, clipcanvaspath**

clipcanvaspath

– **clipcanvaspath** –

Sets the current path to be the clipping path for the current canvas as set by **clipcanvas.**

continueprocess

process **continueprocess** –

Restarts a suspended process.

See also: **suspendprocess, breakpoint**

contrastswithcurrent

color **contrastswithcurrent** boolean

Returns *true* if the *color* argument is different from the current color; otherwise, returns *false* .

This operator takes into account the characteristics of the current device. Boolean operators, such as **eq**, can be used to compare colors without accounting for the current device.

copyarea

dx dy **copyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. The non-zero winding number rule is used to define the inside and outside of the path.

NOTE *This primitive might be used to scroll a text window.*

countfileinputtoken

file **countfileinputtoken** integer

Returns the number of usertokens associated with the given file, ignoring null tokens at the end of the list. (Normally, the returned number is simply the number of user tokens that have been defined, since applications rarely define null user tokens.) The returned index can be used as the next slot into which a user token can be stored.

countinputqueue

– **countinputqueue** num

Returns the number of events currently available from the process' local input queue.

createcolormap

visual **createcolormap** cmap
Returns an empty colormap for the specified visual.

createcolorsegment

cmap color **createcolorsegment** cmapseg
cmap C P **createcolorsegment** cmapsegs
In the first syntactic form, *cmap* is a colormap and *color* is a NeWS color object. The operator returns a single colorsegment of one entry. If the specified colormap is static, the entry returned is the one that has the closest match to the specified color value. If the colormap is dynamic, a new entry is set to the specified color value, unless the colormap is full, in which case the entry returned is the one that most closely matches the specified color.

In the second syntactic form, *cmap* is a colormap; both *C* and *P* are integers. *C* represents the number of colorsegments to be returned. *P* represents the number of planes to be used in the mask of each returned colorsegment.

createdevice

string **createdevice** boolean *or* canvas *or* env
Creates and initializes a new device, such as a framebuffer, keyboard, or mouse. The *string* argument, which is system dependent, indicates the device to be initialized. For example, the strings `/dev/fb`, `/dev/keyboard`, and `/dev/mouse` might represent a framebuffer, keyboard, and mouse.

If **createdevice** fails to create the specified device, it returns *false* . If it succeeds, it returns the specified device. If a framebuffer was specified, the returned device is an object of type **canvas.** If an input device, such as a keyboard or mouse, was specified, the returned device is an object of type **environment**. The returned device is system and implementation dependent.

This operator should only be called during system initialization.

createevent

– **createevent** event
Creates an object of type **event** and initializes its fields to either null or zero.
See also: **awaitevent, redistributeevent , expressinterest , sendevent**

createmonitor

– **createmonitor** monitor
Creates a new monitor object.
See also: **monitorlocked, monitor**

createoverlay

canvas **createoverlay** overlaycanvas
Creates a new canvas that is an *overlay canvas* and is associated with the non-overlay canvas specified by the *canvas* argument.

An overlay canvas can only be created over an existing non-overlay canvas and is always transparent. However, when graphic objects are drawn on an overlay, they appear on the overlay itself, rather than on the canvas below. Overlays are intended for use in transient or animated drawing procedures, such as the creation of *rubber-band* boxes, which expand or contract according to mouse movement, such as when a user is resizing a window.

See Chapter 2, *Canvases* for further information on overlays.

currentautobind

– **currentautobind** boolean

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

NOTE *When the POSTSCRIPT language interpreter encounters an executable name, it searches the dictionary stack from the top to the bottom until it finds a definition for this name. This procedure allows programmers to redefine names selectively; each name can be redefined in a dictionary placed on the dictionary stack above the normal name definition.*

However, the procedure also means that execution time tends to increase in proportion to the size of the dictionary stack. To alleviate this problem, the POSTSCRIPT language provides an operator named **bind** *that circumvents the lookup process. The operator examines the contents of a specified procedure and checks each executable name that it encounters. If a name resolves to an operator object in the context of the current dictionary stack,* **bind** *modifies the procedure by replacing the encountered name with the associated operator object. This has the effect of eliminating the time required by name lookups when the procedure is executed. Note, however, that it also removes the flexibility of being able to change a procedure's behavior by redefining names prior to execution.*

When autobinding is enabled, the effect is as if the **bind** *operator were called automatically in every procedure.*

See also: **setautobind**

currentbackcolor

– **currentbackcolor** color

Returns the background color, which is the color painted by **erasepage**.

See also: **setbackcolor**

currentbackpixel

– **currentbackpixel** integer

Returns an integer that is an index into a colormap and corresponds to the current color of the background.

See also: **setbackpixel**

currentcanvas

– **currentcanvas** canvas

Returns the current value of the canvas parameter in the graphics state.

currentcolor

– **currentcolor** color
Returns the current color as set by **setcolor**, **setrgbcolor**, **sethsbcolor**, or **setpixel**.

currentcursorlocation

– **currentcursorlocation** x y
Returns the position that was occupied by the cursor when the last event was distributed from the global input queue, provided that the event contained meaningful cursor coordinates.
See also: **canvasesunderpoint**

currentfontmem

– **currentfontmem** num
Returns the size of the font memory cache in units of kilobytes: this is the amount of memory that is used to store unused fonts in the system. For an explanation of this cache and its use, see See Chapter 8, *Memory Management.*
See also: **setfontmem**

currentpath

– **currentpath** path
Returns an object of type *path* that describes the current path.

currentpixel

– **currentpixel** integer
Returns an integer that is an index into a colormap and corresponds to the current color of the graphics context.

currentplanemask

– **currentplanemask** integer
Returns the integer currently used as the planemask. The pixel value used by the current graphics context is **AND**'d with the current planemask during drawing operations.
See also: **setplanemask**

currentprintermatch

– **currentprintermatch** boolean
Returns the current value of the **printermatch** flag in the graphics state. The default printermatch state for a process is **true**, consistent with standard POSTSCRIPT language semantics.
See also: **setprintermatch**

currentprocess

– **currentprocess** process
Returns an object that represents the current process.

currentrasteropcode

– **currentrasteropcode** num

Returns a number that represents the current rasterop combination function. See **setrasteropcode** for a table of the rasterop combination functions and a discussion of its use.

See also: **setrasteropcode**

currentstate

– **currentstate** state

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

See also: **setstate**

currenttime

– **currenttime** num

Returns a time value *n.nnn* (in units of 2^{16} milliseconds) that represents time elapsed since some unspecified starting time.

This operator is guaranteed only as follows: the difference of the results of two successive calls is approximately the time that has elapsed between the calls.

currenttimems

– **currenttimems** int

Similar to **currenttime** except returns the time in units of milliseconds and as an integer instead of a real. Note that **currenttime** and **currenttimems** merely return different representations of the same value. **currenttimems** is preferred because it it more accurate.

See also: **currenttime**

damagepath

– **damagepath** –

Sets the current path to be the damage path of the current canvas. The damage path will be cleared.

The *damage path* represents those parts of the canvas that have been damaged and cannot be repainted from stored bitmaps. Processes can arrange to be notified of damage by expressing interest in *damage events*. When damage occurs to a canvas, a damage event is generated by the server.

See also: **clipcanvas**

defaulterroraction

any errorname **defaulterroraction** –

Produces an **$error** dictionary for the current process as if the error specified by *errorname* had been encountered while executing the object *any*. The operator will then execute the **stop** primitive.

NOTE *These actions are similar to the actions of the default error handling procedures described in the PostScript Language Reference Manual.*

deliverevent

? **deliverevent** ?

emptypath

– **emptypath** boolean

Returns *true* if the current path is empty, otherwise *false* .

encodefont

font array **encodefont** font
font name **encodefont** font

If the *array* argument is specified, this operator creates a new font that is identical to the original font specified by the *font* argument, except that the **/Encoding** array of the old font is replaced by the specified *array* argument.

If the *name* argument is specified, the font bearing that name is located in the encoding directory and is encoded.

eoclipcanvas

– **eoclipcanvas** –

This is the same as **clipcanvas**, except that it uses the even-odd rule, rather than the non-zero winding number rule.

See also: **clipcanvas**

eocopyarea

dx dy **eocopyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. The even-odd rule is used to define the inside and outside of the path.

NOTE *This primitive might be used to scroll a text window.*

See also: **copyarea**

eocurrentpath

– **eocurrentpath** path

This is the same as **currentpath**, except that it uses the even-odd rule, rather than the non-zero winding number rule.

eoextenddamage

– **eoextenddamage** –

Adds the current path to the damage shape for the current canvas. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. This operator uses the even-odd rule.

eoextenddamageall

– **eoextenddamageall** –

Adds the visible parts of the current path to the damage shape for the current canvas and the damage shapes of its children. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. The **eoextenddamageall** operator uses the even-odd rule.

eoreshapecanvas

canvas **eoreshapecanvas** –

The **eoreshapecanvas** operator is identical to **reshapecanvas**, except that it uses the even-odd rule to interpret the path.

See also: **reshapecanvas**

eowritecanvas

file *or* string **eowritecanvas** –

This operator is identical to **writecanvas**, except that **eowritecanvas** uses the even-odd rule to define the path.

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

eowritescreen

file *or* string **eowritescreen** –

This operator is identical to **writescreen**, except that **eowritescreen** uses the even-odd rule to define the path.

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

expressinterest

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

Expresses interest in receiving an event distributed from the global event queue. If a *process* argument is specified, interest is expressed on behalf of that process; otherwise, interest is expressed on behalf of the current process.

When passed to **expressinterest**, the *event* becomes an interest, against which each event distributed from the global event queue is compared. When a distributed event matches the interest, a copy of the distributed event is placed on the process' local input queue.

If the *event* argument is already an interest, the **expressinterest** operator takes no action when called.

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

extenddamage

– **extenddamage** –

Adds the current path to the damage shape for the current canvas. If damage was not present on a particular canvas, a *damage* event is sent to processes that have expressed interest. This operator uses the non-zero winding number rule.

extenddamageall

– **extenddamageall** –

Adds the visible parts of the current path to the damage shape for the current canvas and the damage shapes of its children. A *damage* event is distributed if damage was not present on a particular canvas. This operator uses the non-zero winding number rule.

See also: **eoextenddamageall**

file

string1 string2 **file** file

Creates a *file* object for the file identified by *string1*, accessing it as specified by *string2*. This operator is the same as the standard POSTSCRIPT language version, except that a specific search procedure is used to locate existing files. The **file** operator first tries to open *string1* in the current directory (*./string1*). If that fails, it tries to locate and open *string1* in the home directory (*~/string1*). If that fails, it tries to open `$OPENWINHOME/etc/`*string1*.

The **file** operator can be used to create files for connections between client processes and the NeWS server; these files are socket connections and are given special filenames. Files that listen for a connection from some other process have the special filename (`%socketl`*n*), where *n* is the port number that is used for listening. Files that establish a connection between two processes have either the filename (`%socketc`*n*) or the filename (`%socketc`*n.h*), where *n* is the port number and *h* is the hostname. The connection file looks for a listener file on port *n* and host *h* (the default host is the local host); if it finds the specified listener file, it establishes the connection.

findfilefont

string **findfilefont** font

Reads the font family file named by *string* and returns a newly-created font object that refers to it. The font is entered into the **FontDirectory** under the font name in the family file.

NOTE *This operator allows a bitmap font to be loaded after start-up has already occurred.*

fork

proc **fork** process

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

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

getcanvaslocation

canvas **getcanvaslocation** x y

Returns the location of *canvas*, relative to the current canvas. The *x,y* pair is the offset from the origin of the current coordinate system to the origin of *canvas*' default coordinate system.

getcanvasshape

– **getcanvasshape** path

Returns a path object that describes the shape of the current canvas.

See also: **movecanvas**

getcard32

string index **getcard32** integer

Returns an integer that contains the 32 bits in *string*, starting at the 32-bit word offset *index*. Note that this operator has architecture dependencies.

See also: **putcard32**

getcolor

colormapsegment integer **getcolor** color

Returns the color contained in a slot of a colormapsegment. The *colormapsegment* argument specifies the colormapsegment. The *integer* argument specifies the slot number.

See also: **putcolor**

getcompateventdist

– **getcompateventdist** boolean

This operator returns the boolean value of the current process' event synchronization mode.

See also: **setcompateventdist**

getenv

string1 **getenv** string2

Returns the value of the server environment variable *string1*. The value is returned as it exists in the environment of the server process; the value may be modified by **putenv** operations. The **getenv** operator fails with an `undefined` error if *string1* is not present in the environment. The **stopped** operator can be used to recover from the error.

Example

```
{ (ENV) getenv } stopped { pop (default_env_string) } if
```

See also: **putenv**

geteventlogger

– **geteventlogger** process *or* null

Returns the process that is the current event logger, or null if no such process exists.

See also: **seteventlogger**

getfileinputtoken

integer **getfileinputtoken** any
integer file **getfileinputtoken** any

Returns the object associated with the *integer* in *file*'s token list. If no *file* is specified, **currentfile** is used.

getkeyboardtranslation

– **getkeyboardtranslation** bool

Returns *true* if the kernel is interpreting the keyboard, *false* if the task is being performed by POSTSCRIPT language code.

See also: **keyboardtype, setkeyboardtranslation**

getprocesses

– **getprocesses** array

Returns an array of process groups and zombie processes. Each process group is an array of the currently active processes in the process group. Each zombie process is returned as an array containing only the zombie process, since zombie processes are not associated with any process group.

getprocessgroup

process *or* null **getprocessgroup** array

Returns the array of all processes in the process group of either the specified *process* or the current process (if *null* is specified). If *process* is a zombie process, it is the only process in the array, since zombie processes are not associated with any process group.

getsocketlocaladdress

file **getsocketlocaladdress** string

Returns a string that describes the local address of the *file* argument; this argument must be a socket file; normally, it should be a socket that is being listened to.

This operator is generally used by the server to generate a name that can be passed to client programs, telling them how to contact the server. The format of the returned string is unspecified.

getsocketpeername

file **getsocketpeername** string

Returns the name of the host to which *file* is connected. The *file* argument must be an IPC connection to another process. Such files are created with either **acceptconnection** or `(%socket) file`. This operator is normally used with **currentfile** to determine the location from which a client program is contacting the server.

See also: **acceptconnection**

globalroot

– **globalroot** canvas

Returns the global root canvas, which is the root of the server's canvas hierarchy. The global root canvas is a transparent canvas with dimensions 32767 x 32767. Each display screen has a device canvas that is a child of the global root canvas.

harden

any **harden** any

Takes a single argument and returns it unchanged, except that if the argument is a soft reference to an object, a hard reference to the same object is returned.

See also: **soften**

hsbcolor

h s b **hsbcolor** color

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

See also: **rgbcolor**

imagecanvas

canvas **imagecanvas** –

Renders a *canvas* onto the current canvas. This operator is similar to the **image** operator, except that the rendered image comes from a canvas, rather than from a POSTSCRIPT language procedure. When *canvas* is rendered, the unit square is transformed to the same orientation and scale as the unit square in the current transformation matrix.

The current transformation matrix can be modified (using **translate, scale,** or **rotate**) in order to render *canvas* to a particular area within the current canvas.

This operator maps color images onto black and white screens by dithering. The **imagecanvas** operator cannot be used to render a canvas into an overlay.

In the current implementation, the **imagecanvas** and **imagemaskcanvas** operators paint the region within the source canvas' bounding box, rather than painting just the canvas' interior. This difference becomes apparent if you image a non-rectangular canvas.

If you image an unretained canvas that is non-rectangular, the bits outside the canvas' shape but inside the canvas' bounding box are imaged with whatever color they have on the screen. If you image a retained canvas that is not rectangular (either rooted or unrooted), the bits outside the canvas' shape but inside the canvas' bounding box are imaged with whatever color is assigned to 0 (usually white on monochrome screens); these bits were assigned a 0 value when the canvas was made retained.

If you want to omit the area between the canvas' shape and its bounding box, simply clip to the canvas' shape when you image onto the current canvas.

See also: **buildimage, imagemaskcanvas , readcanvas**

imagemaskcanvas

boolean canvas **imagemaskcanvas** –

Renders a *canvas* onto the current canvas. This operator is identical to the **imagemask** operator, except that the image comes from a canvas instead of a POSTSCRIPT language procedure. The *boolean* argument determines whether the polarity of the mask canvas is inverted.

When *canvas* is rendered, the unit square is transformed to the same orientation and scale as the unit square in the current transformation matrix.

The current transformation matrix can be modified (using **translate, scale,** or **rotate**) in order to render *canvas* to a particular area within the current canvas.

See also: **buildimage, imagecanvas , readcanvas**

imagepath

bool canvas **imagepath** –

Makes a one bit deep *canvas* and makes it the current path. The path is suitable for **fill, clip,** or **reshapecanvas,** but not for **stroke.** If *bool* is true, ones in **canvas** are inside the path; if false, ones are outside the path.

insertcanvasabove

canvas x y **insertcanvasabove** –

Inserts the current canvas above *canvas.* The current canvas must be a sibling of *canvas.*

See also: **canvastotop, movecanvas**

insertcanvasbelow

canvas x y **insertcanvasbelow** –

Inserts the current canvas below *canvas.* The current canvas must be a sibling of *canvas.*

See also: **canvastobottom**

isarray?

any **isarray?** bool

Returns **true** if its argument is an array or packedarray, otherwise returns **false.**

keyboardtype

– **keyboardtype** num

Returns a small integer that indicates the kind of keyboard that is attached to the server. The returned *number* is actually the return from the **KIOCTYPE** ioctl, documented under `kb`(4S).

See also: **getkeyboardtranslation, setkeyboardtranslation**

killprocess

process **killprocess** –

Kills *process.*

killprocessgroup

process **killprocessgroup** –

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

See also: **newprocessgroup**

lasteventkeystate

– **lasteventkeystate** array

Returns the **KeyState** key value of the last event delivered by the event distribution mechanism.

lasteventtime

– **lasteventtime** num

Returns the **TimeStamp** key value of the last event delivered by the event distribution mechanism. The value is returned as a time *n.nnn* (in units of $ 2 sup 16 $ milliseconds).

lasteventtimems

– **lasteventtimems** int

Similar to **lasteventtime** except returns the time value in units of milliseconds and as an integer instead of a real. Note that **lasteventtime** and **lasteventtimems** merely return different representations of the same value. **lasteventtimems** is preferred because it it more accurate.

See also: **lasteventtime**

lasteventx

– **lasteventx** num

Returns the *x* coordinate, relative to the current coordinate system, of the last event delivered by the event distribution mechanism.

lasteventy

– **lasteventy** num

Returns the *y* coordinate, relative to the current coordinate system, of the last event delivered by the event distribution mechanism.

localhostnamearray

– **localhostnamearray** array

Returns an array whose first element is the primary hostname of the host on which the server is running, and whose remaining elements (if any exist) are aliases. The value returned by the **localhostname** operator is identical to the first element in the array returned by **localhostnamearray**.

max

a b **max** c

Compares *a* and *b* and leaves the greater of the two on the stack. Works on any data type for which **gt** is defined.

min

a b **min** c

Compares *a* and *b* and leaves the smaller of the two on the stack. Works on any data type for which **gt** is defined.

monitor

monitor procedure **monitor** –

Executes *procedure* with *monitor* locked (entered). At any given time, only one process may have a monitor locked. If a process attempts to lock a locked monitor, the process blocks until the monitor is unlocked. If an error occurs during the execution of *procedure*, and the execution stack is unwound beyond the *monitor*, the *monitor* object becomes unlocked.

See also: **createmonitor, monitorlocked**

monitorlocked

monitor **monitorlocked** boolean

Returns *true* if the *monitor* is currently locked; *false* otherwise.

See also: **createmonitor, monitor**

movecanvas

x y **movecanvas** –
x y canvas **movecanvas** –

If no *canvas* argument is specified, **movecanvas** moves the current canvas to *x,y*, relative to its parent. In this case, *x,y* is an offset from the origin of the parent canvas' default coordinate system to the origin of the current canvas' default coordinate system, measured in units of the current coordinate system.

If a *canvas* argument is specified, **movecanvas** moves *canvas* to *x,y* in the current coordinate system. In this case, *x,y* is an offset from the origin of the current coordinate system to the origin of the repositioned *canvas*' default coordinate system.

See also: **getcanvaslocation**

newcanvas

pcanvas **newcanvas** ncanvas
pcanvas visual cmap **newcanvas** ncanvas

If the *pcanvas* argument alone is specified, the operator creates a new empty canvas, *ncanvas*, whose parent is *pcanvas*. The canvas' coordinate system and shape are undefined until set with **reshapecanvas**.

The *visual* and *colormap* arguments can be used to specify a visual and a colormap for the new canvas. If these arguments are specified, the **Visual** attribute of the specified colormap must match the specified visual. If the arguments are not specified, the canvas' visual and colormap are inherited from its parent.

The new canvas defaults to being opaque if its parent is the framebuffer; transparent otherwise. The canvas defaults to being retained if it is opaque and the number of bits per pixel of the framebuffer is less than the retain threshold.

See also: **reshapecanvas**

newcursor

cursorchar maskchar font **newcursor** cursor
cursorchar maskchar cursorfont maskfont **newcursor** cursor
Creates an object of type **cursor**. Two syntactic forms can be used. With the first, a cursor is constructed using the cursor character *cursorchar* and the mask character *maskchar*; both are selected from *font*. With the second, a cursor is constructed using *cursorchar* from the font *cursorfont* and *maskchar* from the font *maskfont*. In both cases, the new cursor is initialized with a **CursorColor** value of black and a **MaskColor** value of white.

newprocessgroup

– **newprocessgroup** –
Creates a new process group, with the current process as its only member. When a process forks, the child will be in the same process group as its parent.

objectdump

file **objectdump** –
Writes to the specified file a formatted summary of the number of objects that the server has created. Nothing is returned. Note that the specified *file* must be open for writing; otherwise, an `invalidaccess` error is signaled.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

packedarray

objects n **packedarray** packedarray
Creates a packed array object of length *n*. The array contains the specified *objects* as its elements. The operator first removes the non-negative integer *n* from the operand stack. It then removes *n* objects from the operand stack, creates a packed array containing those objects, and puts the resulting packed array object on the operand stack. The resulting object is of type **packedarraytype**, has a literal attribute, and has read-only access. In all other respects, its behavior is identical to that of an ordinary array object.
See also: **currentpacking, setpacking**

pathforallvec

array **pathforallvec** –
The single argument to **pathforallvec** is an array of procedures. The **pathforallvec** operator then enumerates the current path in order, executing one of the procedures in the array for each of the elements in the path. The type of the path element determines which array element will be executed. **moveto, lineto, curveto**, and **closepath**, are array elements 0, 1, 2, and 3, respectively. If the array is too short, **pathforallvec** tries to reduce elements of one type to another. The fifth element is used to handle conic control points. The standard POSTSCRIPT language operator **pathforall** is exactly equivalent to '4 array astore pathforallvec.'

The **pathforallvec** operator should not normally be used: the **pathforall** operator should be used instead.

pause

– **pause** –

Suspends the current process until all other eligible processes have had a chance to execute.

pipe

command **pipe** rfile wfile

Executes the utility whose name is *command.* The input for *command* can be provided with writes on the *wfile* object. The output from *command* can be read from the *rfile* object. The *wfile* object is normally removed from the stack if *command* does not expect input. Popping either object is sufficient to close that portion of the connection. If a two-way, read-write connection is established, it is good practice to close *wfile* before consuming *command*'s output via *rfile* to overcome potential buffering problems.

pointinpath

x y **pointinpath** boolean

Returns *true* if the point *x,y* is inside the current path.

postcrossings

outcanvas incanvas outname inname detailpointer? **postcrossings** –

This operator generates "crossing events", which notify the system of the movement from one canvas to another of a "state"; for example the state can be the canvas under the pointer or the focus. Examples of crossing events are **Enter** events, **Exit** events, and "focus notification" events.

The *outcanvas* argument is the canvas that the state is leaving. The *incanvas* argument is the canvas that the state is entering. Both of these arguments can be specified as either the keyword **/ReDistribute** or `null`; this is useful for managing focus states and other states that need additional modes.

The *outname* argument specifies the **Name** value of the events that indicate the canvas that the state is leaving. The *inname* argument specifies the **Name** value of the events that indicate the canvas that the state is entering. If `null` is specified for either of these arguments, generation of events with that name is suppressed.

The *detailpointer?* argument is a boolean that determines whether or not to generate extra events that indicate the relation of the state holder to the canvas under the pointer. If *detailpointer?* is set to *true* , events with an **Action** value of 5 are delivered to all canvases under the pointer that are also descendants of either *outcanvas* or *incanvas.*

The least common ancestor of *outcanvas* and *incanvas* is determined. Events with **Name** set to *outname* are sent to *outcanvas* and to each of its ancestors up to, but excluding, the least common ancestor; these events are sent in leaf-to-root order. Then, events with **Name** set to *inname* are sent to *incanvas* and to each of its ancestors up to, but excluding, the least common ancestor, in root-to-leaf order.

The **Action** field is set to a value dependent on the canvas' position in the hierarchy with respect to *outcanvas* and *incanvas*, according to the following guidelines:

Table 10-1 *Events sent to* incanvas *and its parents*

Action	*Explanation*
0	The canvas now holds the state; the previous holder was an ancestor of this canvas.
1	The canvas is now an ancestor of the holder of the state; the previous holder was an ancestor of this canvas.
2	The canvas now holds the state; the previous holder was a descendant of this canvas.
3	The canvas now holds the state; the previous holder was not an ancestor or descendant of this canvas.
4	The canvas is now an ancestor of the holder of the state; the previous holder was not an ancestor or descendant of this canvas.
5	The canvas directly or indirectly contains the pointer, and is now a descendant of the holder of the state. The previous holder was not this canvas or an ancestor or descendant of it.
6	The holder of the state is now **ReDistribute**.
7	The holder of the state is now **None**.

Table 10-2 *Events sent to* outcanvas *and its parents*

Action	*Explanation*
0	The canvas used to be the holder of the state; the new holder is an ancestor of this canvas.
1	The holder of the state used to be a descendant of this canvas; the new holder is an ancestor of this canvas.
2	The canvas used to be the holder of the state; the new holder is a descendant of this canvas.
3	The canvas used to be the holder of the state; the new holder is not an ancestor or descendant of this canvas.

Table 10-2 *Events sent to* outcanvas *and its parents— Continued*

Action	*Explanation*
4	The canvas used to be an ancestor of the holder of the state; the new holder is not an ancestor or descendant of this canvas.
5	The canvas directly or indirectly contains the pointer, and used to be a descendant of the holder of the state. The new holder is not this canvas or an ancestor or descendant of it.
6	The holder of the state used to be **ReDistribute.**
7	The holder of the state used to be **None.**

This primitive is provided for the convenience of system event and state managers. An example of postcrossings usage can be found in the X11/NeWS focus manager.

printermatchfont

font boolean **printermatchfont** font

Each font object has a **PrinterMatched** key that determines whether the font is printermatched; the key's value is initialized to the value of the process' printermatch state. The value of a font's **PrinterMatched** key can be retrieved with the **get** operator. The **PrinterMatched** value cannot be set with **put** or **def** because font dictionaries are read-only. The **printermatchfont** operator can be used to construct a new font, with a given **PrinterMatched** value, from an existing font. For example, the following code constructs a Times-Roman font with a false printermatch state:

```
/Times-Roman findfont
 /ISO-Latin-1 encodefont
 12 scalefont
 false printermatchfont
 setfont
```

putcard32

string index integer **putcard32** –

Inserts 32 bits, represented by *integer*, into the value of *string* at the 32-bit word offset specified by *index*. Note that this operator has architecture dependencies.

See also: **getcard32**

putcolor

colormapsegment integer color **putcolor** –

Puts a color into a colormapsegment object. The arguments to **putcolor** are a colormapsegment object (which can be returned by the **createcolorsegment** operation), an integer (which is the number of the colormapsegment slot into which the color is placed), and a color object. If the colormapsegment object has only one slot, the value of the integer argument should be 0. Note that colormapsegment must be writable to use **putcolor**.

See also: **getcolor**

putenv

string1 string2 **putenv** –

Defines the server environment variable *string1* to have the value *string2*. Environment variables inherited by the server may be modified by calls to the **putenv** operator. If the **runprogram** operator is used to create a new UNIX process, the new process inherits the server's environment variables at their current value.

See also: **getenv**

readcanvas

string *or* file **readcanvas** canvas

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

If the specified file cannot be found, an `undefinedfilename` error is generated. If the file cannot be interpreted as a raster file, an `invalidaccess` error is generated.

Note that a canvas read into NeWS with this operator cannot be mapped to the screen. However, the canvas can be used as source for the **imagecanvas** operator. Accessing the irrelevant fields of a canvas created with **readcanvas** does not cause errors, but attempting to execute irrelevant operators (such as **movecanvas** or **reshapecanvas**) will cause error messages.

See also: **imagecanvas, writecanvas**

recallevent

event **recallevent** –

Removes *event* from the global event queue. This primitive can be used to turn off a timer-event that has been sent but not yet delivered.

See also: **sendevent**

redistributeevent

event **redistributeevent** –

Compares *event* against current interests, where *event* is an event object already returned by **awaitevent**. Comparison starts with the interest that immediately follows the successfully matched interest, which previously permitted the event object to be returned by **awaitevent**.

This operator allows an event to be matched with interests that were previously inaccessible (due, for example, to the *exclusivity* of the interest previously matched, or to the *event consumption* previously performed by some canvas).

Note that **redistributeevent** does not reinsert the event into the global event queue. No interest compared with the specified event since the last call to **sendevent** is allowed to match that event again.

See also: **expressinterest**

refcnt

object **refcnt** fixed fixed

Returns two numbers onto the process stack: the first is the total reference count for the specified object; the second is the soft reference count for the specified object. The counts indicate the status of the object after the operator has cleaned up the reference to the object that was given to it on the stack.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

reffinder

object **reffinder** –
object boolean **reffinder** –

Prints to standard output information on all current references to the specified object. The optional *boolean* argument can be *true* (which specifies that only information about hard references is printed) or *false* (which specifies that information about all references is printed).

If the specified object is not a counted type, a message is printed and **reffinder** returns.

This operator does not reside in **systemdict**; it resides in another system dictionary called **debugdict**. To use this operator, you must first place **debugdict** on the dictionary stack by typing **debugdict begin**.

reshapecanvas

canvas **reshapecanvas** –
canvas path width **reshapecanvas** –

If a *canvas* argument alone is specified, this operator sets the shape of *canvas* to be the same as the current path, and sets *canvas*' default transformation matrix to be the same as the current transformation matrix. If *canvas* is the current canvas, an implicit **initmatrix** is performed. The entire contents of the canvas are considered to be damaged. Note that if *canvas* is the current canvas, an implicit **initclip** is performed; the **initclip** operation sets the path to the shape defined by the shape of the current canvas.

The *path* and *width* arguments can only be used if the specified *canvas* is an X canvas; if this is not so, a `typecheck` error is signaled. When the *path* and *width* arguments are specified, **reshapecanvas** can be used to give an X canvas a new border width. The *width* argument is the new border width. The *path* argument represents the drawable part of the X canvas not including the border width; it should be placed in a position inside the current path by a distance equal to *width* pixels. The following code can be used to give an X canvas a new border width.

```
canvas setcanvas
canvas bw-oldbw bw-oldbw width height rectpath
currentpath bw
newpath x y width+2*bw height+2*bw rectpath
reshapecanvas
```

where

bw	=	the new border width
oldbw	=	the old border width
width	=	the new width
height	=	the new height
x y	=	the new x and y location of the canvas in the old coordinate system

Undefined results occur if the width and paths involved are inconsistent, or do not follow the rules for X canvases.

revokeinterest

event **revokeinterest** –
event process **revokeinterest** –
Revokes an interest previously expressed either by the specified *process*, or, if no *process* argument is specified, by the current process. Following execution of this operator, no event matching *event* is distributed to the process. If you specify a *process* argument that is not the same as the value of the interest's **Process** key, you will receive an `invalidacess` error. Likewise, if you specify no *process* argument, but the current process is not the same as the **Process** value of the interest, you will receive an `invalidaccess` error.

Revoking interest on a non-interest has no effect.

See also: **expressinterest**

rgbcolor

r g b **rgbcolor** color
Takes three numbers between 0 and 1, respectively representing the red, green, and blue components of a color, and returns a *color* object that represents the specified color.

runprogram

string **runprogram** –
Forks a UNIX process to execute *string* as a shell command line. Standard input, standard output, and standard error are directed to `/dev/null`.

send

name object **send** –
proc object **send** –
Establishes *object*'s context by putting it and the classes in its inheritance array on the dictionary stack, executes the method, then restores the initial context. In a nested **send**, the previous send context is temporarily removed from the dictionary stack while the nested **send** executes. The *object* argument is the receiver of the message; it can be a class or an instance. In the first form, the *name* argument is the name of the method that is invoked. Any arguments required by the method must be specified; any results of the method are returned.

The second form of **send** uses a *proc* argument instead of the name of a method; *proc* is executed in the context of *object* exactly as if it had been predefined as a method and given a name that was passed to **send**.

See Chapter 5, *Classes*, for more information about classes and the **send** operator.

sendevent

event **sendevent** –
Sends an event into the event distribution mechanism. The event is positioned in the global event queue according to its **TimeStamp**. When the event at the head of the queue has a **TimeStamp** value that is less than or equal to the server's current time, the event is removed from the queue and compared with interests to find matches. Whenever a matching interest is found, the server distributes a copy of the event to the local event queue of the process with the matching interest. The process can then retrieve the event with **awaitevent**.

See Chapter 4, *Events*, for more information about event distribution.
See also: **awaitevent, createevent , recallevent , redistributeevent , expressinterest**

setautobind

boolean **setautobind** –
Enables or disables autobinding for the current process. By default, autobinding is on. (For a description of autobinding, see the entry for the **currentautobind** operator.)
See also: **currentautobind**

setbackcolor

color **setbackcolor** –
Sets the color painted by **erasepage**.
See also: **currentbackcolor**

setbackpixel

pixel **setbackpixel** –
Sets the pixel value of the background to a specified NeWS integer that is an index into a colormap. This color is used by **erasepage**.
See also: **currentbackpixel, setbackcolor**

setcanvas

canvas **setcanvas** –
Sets the current canvas to be *canvas*. Implicitly executes newpath initmatrix initclip.

setcolor

color **setcolor** –
Sets the current color to be *color*. The operation rgbcolor setcolor is identical to **setrgbcolor**; the operation hsbcolor setcolor is identical to **sethsbcolor**. If the requested color is not available in the system, the server will use the closest available color.

setcompateventdist

boolean **setcompateventdist** –
This operator sets the state of the current process' event synchronization mode. When an event is delivered to a process whose synchronization mode is **true**, the server gives the process a chance to run before the next event is removed from the global event queue. The default synchronization state for new processes is **false**; child processes inherit their parents' synchronization state. This operator is provided for backward compatibility with previous versions of the NeWS language. New programs should use the event **Synchronous** key instead.
See also: **getcompateventdist**

setcursorlocation

x y **setcursorlocation** –
Moves the cursor so that its hot spot is at *x*, *y* in the current canvas' coordinate system. Generates an event with **Name** set to **MouseDragged**, **Action** set to null, and the **XLocation** and **YLocation** set to the new cursor location.

seteventlogger

process **seteventlogger** –
Designates a process as an event-logger. The *process* argument must be a process that has expressed an interest (the exact nature of the interest is not significant); the process must also have entered an **awaitevent** loop. Note that the expressed interest must not match any distributed event; the interest is required to prevent **awaitevent** from returning a syntax error. The specified process becomes the *event-logger*. A copy of each event either removed from the global event queue or redistributed with **redistributeevent** will be given to this process before any other (note that the existence of the event-logger does not affect the normal running of the distribution mechanism). When the **awaitevent** loop retrieves the event copies from the event-logger's local input queue, the

event-logger can proceed in whatever way is appropriate. For example, it might print certain key values in a window or to a file.

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

The file `eventlog.ps`, which is described in Chapter 11, *Extensibility through NeWS Procedure Files*, provides a formatted display of events that can be used in the context of the **seteventlogger** operator.

See also: **geteventlogger**

setfileinputtoken

any integer **setfileinputtoken** –
any integer file **setfileinputtoken** –

Takes a specified *object* and *integer* and associates them. The object is placed in *file*'s token list at the index location specified by *integer*. If no file is specifed, **currentfile** (that is, the top file on the execution stack) is used. The *integer* must be between 0 and 65,535. This operator is used to define compressed tokens for communication efficiency.

setfontmem

num **setfontmem** –

Sets the size of the font memory cache. The *num* argument specifies the size of the cache in units of kilobytes. This is the amount of memory that is used to store unused fonts in the system.

See also: **currentfontmem**

setkeyboardtranslation

boolean **setkeyboardtranslation** –

Turns the kernel translation of the keyboard on or off, according to whether the *boolean* argument is specified as *true* or *false*.

See also: **getkeyboardtranslation, keyboardtype**

setpath

path **setpath** –

Sets the current path from the specified path object.

setpixel

integer **setpixel** –

Sets the pixel value of the current graphics context to the specified *integer*.

See also: **currentpixel**

setplanemask

integer **setplanemask** –

Sets planemask to the specified integer. The pixel value used by the current graphics context is **AND**'d with the planemask.

See also: **currentplanemask**

setprintermatch

boolean **setprintermatch** –

Sets the current value of the **printermatch** flag in the graphics state to *boolean*. When printer matching is enabled, text output to the display is forced to be identical to text output to a printer. The metrics used by the printer are imposed on the display fonts (note that this may reduce readability). If printer matching is disabled, readability is maximized; however, the character metrics for the display do not correspond to the printer.

See also: **currentprintermatch**

setrasteropcode

num **setrasteropcode** –

Sets the current rasterop combination function, which will be used in subsequent graphics operations. The **setrasteropcode** operator accepts the following values:

Table 10-3 *Rasterop Code Values*

opcode	*function*
0	0
1	**NOT** (source **OR** destination)
2	(**NOT** source) **AND** destination
3	**NOT** source
4	source **AND** (**NOT** destination)
5	**NOT** destination
6	source **XOR** destination
7	**NOT** (source **AND** destination)
8	source **AND** destination
9	(**NOT** source) **XOR** destination
10	destination
11	(**NOT** source) **OR** destination
12	source
13	source **OR** (**NOT** destination)
14	source **OR** destination
15	**NOT** 0

The default value for the rasterop code is 12.

NOTE *The RasterOp combination function exists only to support emulation of existing window systems. It should not normally be used, since it causes problems when programs are used on a wide range of displays. Currently, the* **image** *primitive does not use the rasteropcode.*

See also: **currentrasteropcode**

setretainthreshold

integer **setretainthreshold** –
After you use this operator, new canvases will be retained by default if its depth is less than or equal to the *integer*. Once the canvas is created, regardless whether it defaults to retained, you can alter its retained state by using the **/Retained** key in the canvas.

setshared

boolean **setshared** –
Depending on the value of *boolean* (`true` or `false`), this operator either enables or disables allocations from the Shared VM pool. When a process is in shared mode, all of its allocations come from the single Shared VM pool in the server. When a process is not in shared mode, all of its allocations come from its own private VM pool. See the information on the **objectdump** operator (in Chapter 8, *Memory Management*) for an account of the types of object allocated in the server. The shared allocation status for newly forked processes is always *false* .

See also: **currentshared**

setstate

graphicsstate **setstate** –
Sets the current graphics state from *graphicsstate.*

See also: **currentstate**

setsysinputtoken

any integer **setsysinputtoken** –
Stores the object *any* into the global system token list at the index *integer*. The global system token list is used by the POSTSCRIPT parser to interpret the compressed POSTSCRIPT byte stream. (The byte stream is used by the `cps` program to reduce network overhead with **cps** clients.) This operator is used at startup in `init.ps` to initialize the global token list.

NOTE *Any changes made to the global token list will drastically affect the operation of any* **cps** *client. It is best not to use* **setsysinputtoken** *unless you understand the byte stream system very well.*

shutdownserver

– **shutdownserver** –
Causes the server to exit.

soft

any **soft** boolean
Takes a single argument and returns *true* if the argument is a soft reference to an object, *false* otherwise.

soften

any **soften** any

The operator takes a single argument and returns it unchanged, except that if the argument is a reference to an object, it is returned as a soft reference. If the operator is used to soften the last existing hard reference to an object, the object becomes obsolete and an *obsolescence event* is generated by the system.

See also: **harden**

startkeyboardandmouse

– **startkeyboardandmouse** –

Initiates server processing of keyboard and mouse input. This operator is called once from the initialization file `init.ps`; it should not be called again.

stoprepeating

– **stoprepeating** –

Clears all state in the repeat key mechanism, stopping all key repeating until another key is held down.

suspendprocess

process **suspendprocess** –

Suspends the given process.

See also: **breakpoint, continueprocess**

tagprint

n **tagprint** –

n file **tagprint** –

Prints the integer *n* (where $-2^{15} \leq n < 2^{15}$) encoded as a tag on the specified output *file*; if no *file* is specified, prints *n* on the current output stream. Tags are used to identify packets sent from the server to client programs. See Chapter 6, *C Client Interface* for information on how the CPS facility uses tags.

truetype

any **truetype** name

Returns a name that identifies the true type of the object *any*. Note that this may be different from the name returned by the **type** operator: this occurs when the specified object is a *magic dictionary* that appears to be a normal dictionary or when a number with a fractional part is represented internally as a scaled integer.

typedprint

object **typedprint** –

object file **typedprint** –

Print the *object* in an encoded form on the specified output *file*; if no *file* is specified, prints *object* on the current output stream. The object can then be read by C client programs, using the CPS facility. The format in which objects are encoded is described in Chapter 6, *C Client Interface*.

unblockinputqueue

– **unblockinputqueue** –

Releases the input queue lock set by **blockinputqueue**. If this reduces the count of locks to 0, distribution of events from the input queue is resumed.

See also: **blockinputqueue**

undef

dictionary key **undef** –

Removes the definition of *key* from the specified *dictionary*.

vmstatus

– **vmstatus** avail used size

Returns three integers, indicating the status of memory usage. The three numbers have different meanings from the Adobe implementation. The value of *avail* is the amount of memory the server has allocated, but is not necessarily using; *used* is the amount of memory currently in use; *size* is the size of the server's heap. All sizes are in units of bytes.

waitprocess

process **waitprocess** value

Waits until *process* completes and returns the value that was on the top of its stack at the time of completion.

See also: **fork**

writecanvas

file *or* string **writecanvas** –

Either opens *string* as a file for writing or, if the argument is a *file*, simply writes to that file. Creates a raster file that contains an image of the region outlined by the current path in the current canvas. The **writecanvas** operator uses the non-zero winding number rule to define the path. If the current path is empty, the whole canvas is written. If the current canvas is partially obscured by one or more canvases that lie on top of it, **writecanvas** writes only the image of the current canvas.

Note that an unretained, rooted canvas must be mapped before using **writecanvas** or **eowritecanvas**. If such a canvas is not mapped, an `invalidaccess` error occurs. Note also that the raster files created by **writecanvas, eowritecanvas, writescreen,** and **eowritescreen** are rectangular. If the canvas that is written to the file is not rectangular, the bits between the canvas' bounding box and the canvas' shape are given 0 values.

Files written by **writecanvas** can be read by **readcanvas**; the file formats that are supported are implementation specific.

See also: **writescreen, eowritescreen , eowritecanvas**

writeobject

file object **writeobject** –
Writes the specified *object* to the specified *file*, in a readable ASCII form.

writescreen

file *or* string **writescreen** –
Either opens *string* as a file for writing or, if the argument is a *file*, simply writes to that file. Creates a raster file that contains a snapshot of the screen, clipped to the current path in the current canvas. The **writescreen** operator uses the non-zero winding number rule to define the path. If the current path is empty, the whole canvas is written. If the current canvas is partially obscured by one or more canvases that lie on top of it, **writecanvas** includes the overlapping canvases in the image.

The **writescreen** operator writes files that **readcanvas** can read; the file formats that are supported are implementation specific.

Example

This operator can be used to do a conventional screen dump, as follows:

```
framebuffer setcanvas (/tmp/snapshot) writescreen
```

See also: **writecanvas, eowritecanvas , eowritescreen**

11

Extensibility through NeWS Procedure Files

In addition to operator and type extensions, which are part of the server itself, NeWS also supplies various POSTSCRIPT files that provide support for the NeWS programming environment; the files are loaded automatically when NeWS is initialized. You can examine these files and modify the procedures that they contain. However, if you modify them, portable NeWS programs may not run on your server.

This chapter gives an overview of the POSTSCRIPT extension files.

11.1. Initialization Files

When the NeWS server is initialized, the following extension files are automatically loaded:

`init.ps`
Initializes the frame buffer, loads most of the other initialization files described in this section, defines and starts the server, and sets various constants and system-defaults.

`redbook.ps`
Defines some POSTSCRIPT language operators that are in the *PostScript Language Reference Manual* but are not NeWS primitives.

`basics.ps`
Defines the utilities necessary to run NeWS as a filter.

`cursor.ps`
Builds a dictionary useful for naming characters in **cursorfont**, which is a special font of cursors. Client-defined cursor fonts can also be built.

`statdict.ps`
Adds the **statusdict** to the **systemdict** for users needing extreme printer compatibility. The **statusdict** dictionary contains printer-specific operators such as **printername** and **setsccbatch**, as specified in Section D.6 of the *PostScript Language Reference Manual.* Many of these operators are pseudo-implemented, since they have no meaning in a window system. The file `statusdict.ps` is loaded automatically by `init.ps` at start-up.

NOTE *NeWS contains many extensions to POSTSCRIPT that do not work on printers. If you have code that you wish to send both to a NeWS server and to a printer, you should test whether the* **newcanvas** *primitive is in* **systemdict**, *since only NeWS servers have such an operator defined.*

`compat.ps`

Defines routines that make the server backwards-compatible with older NeWS client programs; in effect, the server is programmed to emulate previous versions of itself.

`util.ps`

Simple utilities shared by packages and NeWS applications; anything that is used by more than one package should be defined in here.

`class.ps`

Implements the NeWS class mechanism and the methods supplied by the base **Object** class.

NOTE *Some class operators are implemented in C for performance reasons; however, definitions of them in the POSTSCRIPT language are still provided.*

11.2. User-Created Extension Files

This section describes files that can be created by the user; these files can contain customized NeWS initialization procedures. When NeWS is initialized, `init.ps` automatically searches for these files. The search begins in the directory from which NeWS was started; if the files are not found, the directories `~/` and `$OPENWINHOME/etc/` are searched in turn.

`.startup.ps`

Contains code fragments created by the user. If `.startup.ps` exists, its contents are executed by `init.ps` before any other package is loaded.

`.user.ps`

Contains the user's own definitions of POSTSCRIPT operators, including redefinitions of operators already in NeWS. If `.user.ps` exists, its contents are executed by `init.ps` after all other packages are loaded.

Other Extension Files

The following files all define extensions to NeWS; the files are loaded by individual programs rather than by `init.ps`:

`debug.ps`

Contains POSTSCRIPT procedures used for debugging.

`eventlog.ps`

Contains a small package for monitoring event distribution, described under Section 11.15, *Logging Events* below.

`journal.ps`

Contains a package for recording user actions and replaying them in *player-piano* mode, described below in Section 11.11, *Journalling Utilities*.

`repeat.ps`

Implements variable-rate repeating on keyboard keys. See Section 11.14, *Repeating Keys* below.

Extension File Contents

The following sections describe some of the most useful POSTSCRIPT procedures that are contained by the files that have been described above. Note that more exist than are documented here. All of the procedures can be customized to suit the user's individual needs.

11.3. Miscellaneous Utilities

The following utilities provide miscellaneous functionality:

append

obj1 obj2 **append** obj3

Concatenates arrays, strings, and dictionaries. In the case of duplicate dictionary keys, the keys in the second dictionary overwrite those of the first.

buildsend

name *or* array object **buildsend** proc

Builds a procedure that sends a message to *object*. This procedure is self-contained and does not depend on being in a certain dictionary context. This is useful for callback procedures such as the following:

```
/myinstance /new MyClass send def
/dosomething myinstance buildsend /installcallback Manager send
```

case

value {key proc key key proc...} **case** –

Compares *value* against several keys, performing the associated procedure if a match is found. The key **/Default** matches all values.

The following example uses **case** to convert a number to a string:

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

cleanoutdict

dict **cleanoutdict** –

Undefines every key in the dictionary *dict* using **undef**.

createcanvas

parentcanvas width height **createcanvas** canvas

Creates *canvas*, a child of *parentcanvas*, located at (0, 0) relative to its parent and possessing the given *width* and *height*, both of which are numbers.

createdevicecanvas

string **createdevicecanvas** canvas *or* boolean

Creates and initializes additional framebuffer canvases. The *string* argument, which is system dependent, indicates the display device that is to be initialized (for example, `/dev/cgtwo0`).

If **createdevicecanvas** fails to create the specified framebuffer canvas, it returns **false**. If it succeeds, it returns the framebuffer canvas.

This operator should only be called as part of system initialization (for example, from a `.startup.ps` file).

currentshared

– **currentshared** boolean
Returns *true* or *false* , depending on whether the current allocation status of the current process from the shared VM pool is enabled or disabled. See **setshared** for an explanation of the shared VM pool.

See also: **setshared**

cvad

array **cvad** dict
Considers *array* as a list of key-value pairs and fills them into a newly-created dictionary.

cvas

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

cvis

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

dictbegin

– **dictbegin** –
Combined with **dictend,** creates a dictionary large enough for subsequent **defs** and puts it on the dictionary stack. This lets you avoid needing to guess the size of the dictionary to be created.

dictend

– **dictend** dict
Returns the dictionary created by a previous **dictbegin**; together, they "shrink-wrap" a dictionary around your **def** statements.

The following example demonstrates how to use the **dictbegin** and **dictend** pair:

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

See also: **dictbegin**

dictkey

dict value **dictkey** key true *or* false
Searches *dict* for *value* and returns the corresponding key, if it is present. If *value* corresponds to several keys within *dict*, only one of the keys is returned.

fontascent

font **fontascent** number

Returns the specified *font*'s ascent, which is the logical distance that a character in the font extends above the baseline. Specific characters may extend beyond this distance. The measurement is given in units of the current coordinate system.

fontdescent

font **fontdescent** number

Returns *font*'s descent (as a positive number), which is the logical distance that a character in the font extends below the baseline. Specific characters may extend beyond this distance. The measurement is given in units of the current coordinate system.

fontheight

font **fontheight** number

Returns *font*'s height, which is the sum of **fontascent** and **fontdescent**.

fprintf

file formatstring argarray **fprintf** –

Prints to *file*.

The following example prints the amount of time during which the NeWS server has been running on your console:

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

See also: **console**

growabledict

– **growabledict** dict

Creates a large, growable dict and leaves it on the operand stack. The dictionary has a maximum size of 5000 key/value pairs. This large size limit is the only difference between a dictionary created with **growabledict** and a dictionary created with the POSTSCRIPT language **dict** operator.

litstring

str **litstring** str'

Replaces escapes in strings with escaped escapes.

The following example produces the string (\(blank\\n\)):

```
(blank\n) litstring
```

modifyproc

proc {head} {tail} **modifyproc** {head proc tail}

Adds a *head* and/or a *tail* modification to a procedure, leaving on the stack an executable array that contains the modified procedure body. You can use this to override the behavior of a procedure.

The following code modifies the existing version of 'myproc' by prepending the sequence '(myproc called\n) print' to the contents of the procedure each time it is invoked:

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

NOTE *You can use a literal name in place of any procedure you give to* **modifyproc**. *If this name is associated with a procedure in the current dictionary context, this procedure will be used in its place.*

nulloutdict

dict **nulloutdict** –

Defines every key in the dictionary *dict* to be **null**.

printf

formatstring argarray **printf** –

This is the printing form of **sprintf**. Prints on the standard output file, like **print**.

See also: **dbgprintf**

random

– **random** num

Returns a random number in the range [0,1].

RGBcolor

red green blue **RGBcolor** color

Converts color values, specified by *red*, *green*, and *blue* values between 0 and 255, into a NeWS color object.

See also: **rgbcolor**, **setcolor**

refork

processname proc **refork** –

Check to see whether a process specified by *processname* is running. If so, that process is killed with **killprocess**. Then the process (*proc*) is forked.

sendstack

– **sendstack** array

Returns the current send stack as an *array*.

sleep

interval **sleep** –
sleep causes the current process to sleep for *inerval* minutes. The following example causes the current process to sleep for one second:

```
1 60 div sleep      % sleep for one second
```

sprintf

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

An example is given below:

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

The above example puts the following string on the stack:

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

stringbbox

string **stringbbox** x y w h
Returns *string*'s bounding box.
See also: **fontascent, fontdescent , fontheight**

11.4. Array Utilities

The following utilities are provided to perform operations on arrays.

arraycontains?

array value **arraycontains?** bool
Returns the boolean true if the indicated *value* is found in the specified *array.*

arraydelete

array index **arraydelete** –
Returns a new array, deleting the value in array at position *index.* If *index* is beyond the end of the array, the last item in the newly-constructed array is deleted. Thus:

[/a /b 0 /x /y] 2 arraydelete ⇒ [/a /b /x /y]

arrayindex

array value **arrayindex** index boolean
Given an *array* and specified *value*, **arrayindex** returns the *index* of the value (if it is found) and the boolean **true**; if the *index* is not found, the operator returns no index value and the boolean **false**.

arrayinsert

array index value **arrayinsert** newarray

Creates a new array one larger than the initial array by inserting *value* at position *index*. If *index* is beyond the end of the array, *value* is appended to the end of the array. Thus:

```
[/a /b /x /y] 2 0 arrayinsert ⇒ [/a /b 0 /x /y]
```

arrayop

A B proc **arrayop** C

Performs *proc* on pairs of elements from arrays *A* and *B* in turn (for the union of the set), placing the result in array *C*.

Two examples are given below:

```
    [1 2 3] [4 5 6] {add} arrayop ⇒ [5 7 9]
and
    [3 4 5] [4 5 6 6] {add} arrayop ⇒ [7 9 11]
```

arrayreverse

array **arrayreverse** array_reversed

Reverses the elements of the specified *array*.

An example is given below:

```
[3 56 7 8 2 1] arrayreverse
```

This example produces the following result:

```
[1 2 8 7 56 3]
```

arrayreverseFast

array **arrayreverseFast** array_reversed

Reverses the elements of the specified *array*. It runs more quickly than **arrayreverse** but uses the operand stack; thus, it may result in stackoverflow errors.

NOTE *Stack usage is twice the array size.*

arraysequal?

array_A array_B **arraysequal?** bool

Compares the contents of the two arrays. If they are equal, it returns true, otherwise it returns false.

currentpacking

– **currentpacking** bool

Returns the current array-packing mode.

See also: **packedarray, setpacking**

isarray?

any **isarray?** boolean

Returns a boolean indicating whether the object is one of the array types.

quicksort

array proc **quicksort** array

Uses the process *proc* as a rule to sort the contents of the *array*.

An example is given below:

```
[7 9 8 3 4] {gt} quicksort
```

This example produces the following result:

```
[3 4 7 8 9]
```

setpacking

boolean **setpacking** –

Sets the array packing mode to the specified boolean value. This determines the type of executable arrays subsequently created by the POSTSCRIPT language scanner. If the specified *boolean* is *true* , packed arrays are created; if the *boolean* is *false* , ordinary arrays are created.

The array-packing mode affects the creation of procedures by the scanner when program text bracketed by { and } is encountered in the following circumstances:

- During interpretation of an executable file or string object
- During execution of the **token** operator

Note that it does not affect the creation of literal arrays by the [and] operators or by the **array** operator.

The setting continues to exist until it is overriden by a further call to **setpacking**, or undone by a call to **restore**. The packing mode is set on a per-process basis. A child process inherits the packing mode of its parent.

See also: **currentpacking, packedarray**

11.5. Conditional Utilities

The following utilities are provided to allow you to specify conditional operations where a value may or may not already be defined.

?get

dict key default **?get** value
If the specified *key* is found in the *dict*, its value is returned on the stack; otherwise the *default* value is returned on the stack.

?getenv

envstr defaultstr **?getenv** str
Returns the specified *envstr* as a string on the stack, if it differs from the specified *defaultstr*.

?load

key default **?load** value
Searches for the specified *key* through the dictionary stack, starting with the topmost dictionary. If the key is found, the *value* is returned on the stack; otherwise the *default* value is returned on the stack.

?put

dict key value **?put** –
Check if the *key-value* pair exists in the *dict*. If not, add the pair to the dictionary.

?undef

dict key **?undef** –
Remove the specified *key* from the dictionary, if the key is present.

11.6. Input Utilities

The following utilities provide functionality in the area of input and event management.

eventmgrinterest

eventname eventproc action canvas **eventmgrinterest** interest
Makes an interest that is suitable for use by **forkeventmgr** or **expressinterest**.

The following example creates an event manager that handles popping up a menu.

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

forkeventmgr

interests **forkeventmgr** process
Forks a process that expresses interest in *interests*, which may be either an array whose elements are interests or a dictionary whose values are interests. Each interest must have an executable match that consumes the event returned by **awaitevent** (**eventmgrinterest** produces interests of this type)

The following example of **forkeventmgr** forks an event manager to watch for a /DownTransition of the MenuButton.

```
/MyEventMgr [
   MenuButton                    % event_name
   {/popup MyMenu send}          % event_proc
   /DownTransition               % action
   MyCanvas                      % canvas
   eventmgrinterest              % build an interest
] forkeventmgr def
```

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

getanimated

x0 y0 procedure **getanimated** process

Forks a process that does animation while tracking the mouse, returning the process object *process* to the parent process. Each time the mouse moves, the process executes 'erasepage x0 y0 moveto,' pushes the current mouse coordinates *x* and *y* onto its stack, and calls *procedure*. The variables *x0*, *y0*, *x*, and *y* are available to *procedure*. After *procedure* returns, the process executes the **stroke** operator. Thus, *procedure* can use *x0*, *y0*, *x*, and *y* to build a path that is drawn each time the mouse is moved — drawing a line to the current cursor location, for example. (Note that this routine is typically useful only when the current canvas is an overlay canvas.)

The process calling *procedure* exits when the user clicks the mouse; it leaves the final mouse coordinates in an array '[x y]' on top of its stack, so that they are available to the parent process via the **waitprocess** operator. Since **erasepage** is executed each time the mouse is moved, the current canvas should be an overlay canvas when you call **getanimated**. **getanimated** is used to implement most rubber-banding operations on the screen such as in the `rubber` demo program.

See also: **createoverlay, waitprocess**

getclick

– **getclick** x0 y0

Uses **getanimated** to let the user indicate a point on the screen. **getclick** returns the location of the click on the stack.

getrect

x0 y0 **getrect** process

Uses **getanimated** to let the user "rubber-band" a rectangle with a fixed origin *x0*, *y0*. Returns a process with which you can retrieve the coordinates of the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x1 y1] (in an array) on the stack.

The following example sizes a window:

```
100 100 getrect waitprocess
```

This example produces the following result:

[400 432]

See also: **waitprocess**

getwholerect

– **getwholerect** process

Uses **getclick** and **getrect** to let the user indicate both the origin and a corner of a rectangle. Returns a process with which you can retrieve the coordinates of both the origin and the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x0 y0 x1 y1] (in an array) on the stack.

?revokeinterest

event **?revokeinterest** –

Revokes interest in an event. This operator is identical to **revokeinterest,** except that it does not generate `invalidaccesserrors` if the interest has already been revoked.

setstandardcursor

primary mask canvas **setstandardcursor** -

Sets *canvas*'s cursor to the cursor composed of the *primary* and *mask* keywords. *primary* and *mask* must be cursors in **cursorfont**, the font of standard system cursors loaded by `cursor.ps`.

The following example sets the cursor in 'MyCanvas' to an hourglass to indicate that its process will not be responding to user-input for a while.

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

The following table represents the cursors and their masks in **cursorfont:**

Table 11-1 *Standard NeWS Cursors*

Primary Image	*Mask Image*	*Description*	*When/Where Used*
basic	**basic_m**	Left pointing arrow	OPEN LOOK default
move	**move_m**	Move pointer	OPEN LOOK object movement
copy	**copy_m**	Duplicate pointer	OPEN LOOK object copying
busy	**busy_m**	Stopwatch	OPEN LOOK application busy
ptr	**ptr_m**	arrow pointing to upper left	Lite default cursor
beye	**beye_m**	bullseye	Lite window frame
rtarr	**rtarr_m**	"→" arrow	Lite menus
xhair	**xhair_m**	crosshairs ("+" shape)	
xcurs	**xcurs_m**	"×" shape	Lite icons
hourg	**hourg_m**	hourglass shape	start-up/Lite canvas busy
nouse	**nouse_m**	no cursor	
stop	**stop_m**	Stop sign	

11.7. Rectangle Utilities

The following operators manage rectangular coordinates and paths; other graphics procedures are listed below, under *Graphics Utilities*.

insetrect

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

points2rect

x y x' y' **points2rect** x y width height
Converts a rectangle specified by any two opposite corners to one specified by an origin and size.

rect

width height **rect** –
Adds a rectangle to the current path at the current pen location.

rectpath

x y width height **rectpath** –
Adds a rectangle to the current path with *x,y* as the origin.

rectsoverlap

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

rect2points

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

11.8. Graphics Utilities

The following operators can be used to create graphics in canvases.

colorhsb

color **colorhsb** h s b
Returns the HSB values for the given color.

colorrgb

color **colorrgb** r g b
Returns the RGB values for the given color.

cshow

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

fillcanvas

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

insetrrect

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

ovalframe

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

ovalpath

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

polyline

array **polyline** –
Draws lines using numbers from *array*. Considers *array* as an array of (*dx,dy*) pairs and then executes *dx dy* **rlineto** for each pair.

polypath

x y array **polypath** –
Starts a path at (*x,y*) and then draws lines using *array* as for **polyline**. Closes the path at the end.

polyrectline

array **polyrectline** –
Draws rectlinear lines using numbers from *array*. If *array* contains [*a0 a1 a2* ...], this does the equivalent of *a0* 0 **rlineto** 0 *a1* **rlineto** *a2* 0 **rlineto** and so forth.

polyrectpath

x y array **polyrectpath** –
Starts a path at (*x*,*y*) and draws rectlinear lines as for **polyrectline**. Closes the path at the end.

rectframe

thickness x y w h **rectframe** –
Creates a path composed of two rectangles, the first with origin *x*,*y* and size *w*,*h*; the second inset from this by *thickness*. Calling **eofill** fills the frame, while **stroke** creates a "wire frame" around it.

rrectframe

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

rrectpath

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

rshow

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

setshade

gray *or* color **setshade** –
Sets the current color to *gray* or to *color* value. The argument may be either a color or a shade of gray.

strokecanvas

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

11.9. File Access Utilities

The following operators provide file access functionality:

DefineAutoLoads

array **DefineAutoLoads** –
NeWS defines many operators that may never be used. To avoid loading the POSTSCRIPT code definition of every NeWS object at initialization, you can "lazy-define" an object to be the action of loading a file. When the object is first accessed, the file is read in; the loaded file should normally redefine the object to its original value. This form of definition is especially useful for classes, since all the methods and utility procedures that use a class can be defined in a single file, which is only read in when the class is first used. **DefineAutoLoads** takes an array of object-filename pairs.

filepathopen

filename patharray access **filepathopen** path file true *or* name false
Takes a filename, an array of path strings (such as those produced by **filepath-parse**) and the same access control string as **file**; it then tries to open *filename* in each of the paths in turn. As soon as it succeeds, it returns three values: the path that successfully located the file, a file object, and **true**. If it fails, it returns two values: *filename* and **false**.

filepathparse

pathstring **filepathparse** patharray
Takes a colon-separated set of pathnames and returns the pathnames as an array of strings.

The following code parses the pathnames in the environment variable MYPATH; if the variable does not exist, a default set of strings is loaded.

```
/mypath (MYPATH) (.:~/bin:$OPENWINHOME/bin) ?getenv filepathparse def
```

filepathrun

filename patharray **filepathrun** path true *or* name|path false
Takes a filename and an array of path strings; the operator attempts to run the resulting file. If it cannot find the desired file in any of the given paths (using **filepathopen**), it returns *filename* and **false**. If it succeeds in finding the file, it runs the file (using **cvx exec**) in a stopped environment and reports errors. If it finds the file but cannot access or run it, **filepathrun** leaves the full path to the file and **false** on the stack. It also checks whether the file left anything on the execution stack and prints an error if this occurred. If no file was left on the stack, **filepathrun** leaves the full path to the file and **true** on the stack.

LoadFile

string **LoadFile** boolean
This is a robust, more general version of **run**. It is used to execute most NeWS startup files, returning **false** if it has problems, **true** otherwise. It searches for files in several locations: first, it prepends the user's home directory and tries to read from there; then, it passes the actual filename *string* to **file**. **file** looks first in the directory in which the X11/NeWS server was initialized, then in $OPENWINHOME/etc.

11.10. CID Utilities

The POSTSCRIPT files supplied by NeWS include a simple CID (Client IDentifier) synchronizer package. This generates a unique identifier used to generate a channel for client communication.

cidinterest

id **cidinterest** interest
Creates an interest appropriate for use with **forkeventmgr**. The callback procedure installed in this interest simply executes the code fragment stored in the event's **/ClientData** field.

cidinterest1only

id **cidinterest1only** interest
This is a special form of **cidinterest** that processes only one code fragment. It automatically **exits** by itself, rather than requiring the client to send the **exit**.

sendcidevent

id proc **sendcidevent** –
Sends a code fragment to a process created by the **cidinterest** – **forkeventmgr** usage shown above.

uniquecid

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

11.11. Journalling Utilities

The following utilites allow you to control the journalling mechanism. With this mechanism, you can record and play back NeWS user input events. The file `$OPENWINHOME/demo/journaldemo` implements the following three procedures:

journalplay

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

journalrecord

– **journalrecord** –
Starts a journalling session by opening the journalling file and logging user actions to it. The default filename is `/tmp/NeWS.journal`.

journalend

– **journalend** –
Ends a journalling session started by **journalrecord** and closes the journalling file.

Only raw mouse and keyboard events are replayed; thus, the system should be in exactly the same state at the beginning of the replay as it was at the start of the journalling session; this means that the same windows should exist in the same positions on the screen, the same user should be running the system from the same directory, and so forth. The **journalplay** operator automatically repositions the mouse to the exact position it occupied at the start of the session.

Journalling Internal Variables

The journalling utilities use the following internal variables:

- **RecordFile** — the journalling file.
- **PlayBackFile** — initially identical to **RecordFile**, this is the file from which playback takes place.

- **PlayForever** — play forever if true.
- **State** — the current state of journalling system.

These variables are explained more fully in the comments of the file `$OPENWINHOME/demo/journaldemo`. They are defined in the NeWS dictionary **journal**, created in **systemdict.**

11.12. Constants

The following constants and environment variables are provided:

console

– **console** file
Returns the file object for the system's console. Use with **fprintf** to write messages to the console.
See also: **fprintf**

fontpath

– **fontpath** string
Puts on the stack the directory search path NeWS uses to find fonts.

framebuffer

– **framebuffer** canvas
Returns the framebuffer canvas that currently contains the mouse pointer. When a client first connects to the server, the value of the **framebuffer** variable is copied from the server's **systemdict** to the **userdict** of the client connection process. The client should usually use this **framebuffer** value as the parent of its top-level application canvases.

localhostname

– **localhostname** string
Returns the network hostname of the host on which the server is running.

minim

– **minim** real
Returns the smallest value that is representable in NeWS, which is 2^{-16}.

nulldict

– **nulldict** dict
Returns an empty, zero-length dictionary.

nullproc

– **nullproc** procedure
Returns a no-op procedure. Equivalent to { }.

nullstring

– **nullstring** string
Returns an empty string. Equivalent to ( ).

openwinversion

– **openwinversion** string
Puts the full version number of the OpenWindows system on the stack.

version

– **version** string
Puts the version of the NeWS system on the stack.

HOME

– **HOME** string
Puts the absolute pathname to the user's home directory on the stack, or '.' if the `HOME` environment variable is not set.

OPENWINHOME

– **OPENWINHOME** string
Puts the pathname in the `OPENWINHOME` environment variable on the stack, or `/home/openwin` if this is not set. This operator is used to locate NeWS files; users should set this environment variable if they install NeWS in a non-standard location.

11.13. Key Mapping Utilities

A key may be unbound using the **unbindkey** procedure. The **bindkey** and **unbindkey** operators are described below.

bindkey

key arg **bindkey** –
Creates a new process that waits for *key* to be pressed and executes *arg* whenever that happens. If *arg* is an executable array, name, or string, it is simply handed to the PostScript interpreter. Otherwise, if it is a string, the following expression is evaluated:

{ *arg* runprogram }

The following example binds the string `!make` to key [F8] and assigns the NeWS-SunView selection converters to [F9] and [F10][6]:

[6] The [F10] function key doesn't exist on Sun-3 keyboards.

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

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

unbindkey

key arg **unbindkey** –

Removes the binding of the specified *key* (there is no need to call **unbindkey** before rebinding a key to a new value; the new value replaces the old in **bindkey**).

The following example unbinds the key that was bound in the previous example:

```
/FunctionF9 unbindkey
```

11.14. Repeating Keys

By default, the keys of the standard typing array (which does not include the function or shift keys) repeat 20 times per second, after a .5 second threshold. The repeating keys behavior is implemented by a standalone repeat-keys package, `$OPENWINHOME/lib/NeWS/repeat.ps`, which is loaded as part of the *extended input system* started by `init.ps`. You can adjust the threshold and repeat rates according to your preference; you can do this by modifying within your `.startup.ps` file the **KeyRepeatThresh** and **KeyRepeatTime** keys of your **UserProfile** dictionary. This is demonstrated by the following example:

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

11.15. Logging Events

The file `eventlog.ps` defines a procedure (**eventlog**) to turn logging of event distribution on and off, and a dictionary (**UnloggedEvents**), which defines those events to be excluded from the log record. "Logging" means that a copy of each event is printed as it is taken out of the event queue for distribution. This is useful for debugging the server and for clients that use events heavily. The fields of the event that are printed are **Serial, TimeStamp, Location, Name, Action, Canvas, Process, KeyState,** and **ClientData.**

The **Journal** application uses the event logging mechanism to allow the user to record and play back user actions. See the **journalling(1)** manual page for more

information.

eventlog

bool **eventlog** –
Turns event logging on if the boolean is **true**, off if it is **false**.

The following example shows a typical log message:

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

Log messages are sent to standard output (event logging uses the POSTSCRIPT operators **print** and ==).

UnloggedEvents

This is a dictionary of event names that are specified by the user; NeWS does not log these events. The default definition of **UnloggedEvents** is as follows:

```
/UnloggedEvents 20 dict dup begin
   /Damaged dup def
   /MouseDragged dup def
end def
```

A

NeWS Operators

This appendix lists all the current NeWS operators, alphabetically first, then by type.

A.1. NeWS Operators, Alphabetically

listenfile	**acceptconnection** file	listens for connection
num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
boolean errorname	**assert** –	generates an error
–	**awaitevent** event	blocks for event
–	**beep** –	generates audible signal
num	**blockinputqueue** –	blocks input events
–	**breakpoint** –	suspends current process
width height bits/sample matrix proc	**buildimage** canvas	constructs canvas object
–	**canvasesunderpath** array	returns canvases under path
x y *or* null	**canvasesunderpoint** array	returns canvases under point
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clearsendcontexts** –	removes history of send contexts
–	**clipcanvas** –	clips to canvas boundary
–	**clipcanvaspath** –	sets current path to clip
process	**continueprocess** –	restarts suspended process
color	**contrastswithcurrent** boolean	compares colors
dx dy	**copyarea** –	copies current path to *dx, dy*
file	**countfileinputtoken** integer	returns associated usertokens
–	**countinputqueue** num	returns count of input queue
visual	**createcolormap** cmap	returns colormap for visual
cmap color	**createcolorsegment** cmapseg	returns colorsegment
cmap C P	**createcolorsegment** cmapsegs	returns colorsegments
string	**createdevice** boolean, canvas *or* env	creates canvas or environment object
–	**createevent** event	creates event
–	**createmonitor** monitor	creates monitor object
canvas	**createoverlay** overlaycanvas	creates overlay canvas
–	**currentautobind** boolean	tests whether autobinding is on
–	**currentbackcolor** color	gets color painted by **erasepage**

–	**currentbackpixel** integer	returns background pixel
–	**currentcanvas** canvas	returns current canvas
–	**currentcolor** color	returns current color
–	**currentcursorlocation** x y	returns mouse coordinates
–	**currentfontmem** num	returns size of font memory cache
–	**currentpath** path	returns current path
–	**currentpixel** integer	returns index of current pixel
–	**currentplanemask** integer	returns current planemask
–	**currentprintermatch** boolean	returns **printermatch** value
–	**currentprocess** process	returns current process
–	**currentrasteropcode** num	rasterop combination function
–	**currentshared** boolean	tests whether allocation status is enabled
–	**currentstate** state	returns **graphicsstate** object
–	**currenttime** num	returns current time value
–	**damagepath** –	sets path to damage path
–	**defaulterroraction** –	produces **$error** dictionary for process
–	**emptypath** boolean	tests current path
font array	**encodefont** font	duplicates font using new encoding
font name	**encodefont** font	encodes font
–	**eoclipcanvas** –	clips to current canvas
dx dy	**eocopyarea** –	copies area to *dx, dy*
–	**eoextenddamage** –	extends damage path
–	**eoextenddamageall** –	extends damage path to all
canvas	**eoreshapecanvas** –	reshapes canvas
file *or* string	**eowritecanvas** –	writes canvas to file
file *or* string	**eowritescreen** –	writes screen to file
event	**expressinterest** –	enables reception of events
event process	**expressinterest** –	enables reception of events
–	**extenddamage** –	extends damage path
–	**extenddamgeall** –	extends damage path
string1 string2	**file** file	creates file object
string	**findfilefont** font	reads font family file, returns font
proc	**fork** process	creates new process
canvas	**getcanvaslocation** x y	returns canvas location
–	**getcanvasshape** path	returns path object of canvas shape
string index	**getcard32** integer	returns bits from offset
colormapsegment integer	**getcolor** color	returns color from colormapsegment
string1	**getenv** string2	gets value of *string1* in server
–	**geteventlogger** process	gets event logger process
integer	**getfileinputtoken** any	returns file input token
integer file	**getfileinputtoken** any	returns file input token
–	**getkeyboardtranslation** boolean	returns mode of translation
–	**getprocesses** array	returns array of process groups
process *or* null	**getprocessgroup** array	returns array of processes
file	**getsocketlocaladdress** string	returns address of file
file	**getsocketpeername** string	returns name of host connected
–	**globalroot** canvas	returns the global root canvas
any	**harden** any	returns reference as hard reference
h s b	**hsbcolor** color	returns color matching ***h s b***

Operands	Operator / Results	Description
canvas	**imagecanvas** –	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas** –	analogous to **imagemask**
canvas x y	**insertcanvasabove** –	inserts above current canvas
canvas x y	**insertcanvasbelow** –	inserts below current canvas
–	**keyboardtype** num	returns type of keyboard
process	**killprocess** –	kills process
process	**killprocessgroup** –	kills process group
–	**lasteventkeystate** array	returns **KeyState**
–	**lasteventtime** num	returns **TimeStamp**
–	**lasteventx** num	returns *x* coordinatate of event
–	**lasteventy** num	returns *y* coordinatate of event
–	**localhostnamearray** array	returns network hostname and aliases
a b	**max** c	leaves maximum on stack
a b	**min** c	leaves minimum on stack
monitor procedure	**monitor** –	executes procedure with locked monitor
monitor	**monitorlocked** boolean	checks state of monitor
x y	**movecanvas** –	moves canvas to *x y*
x y canvas	**movecanvas** –	moves canvas to *x y*
pcanvas	**newcanvas** ncanvas	creates new canvas
pcanvas visual cmap	**newcanvas** ncanvas	creates new canvas
cursorchar maskchar font	**newcursor** cursor	creates cursor
cursorchar maskchar cursorfont maskfont	**newcursor** cursor	creates cursor
–	**newprocessgroup** –	creates new process group
file	**objectdump** –	writes summary of created objects
objects n	**packedarray** packedarray	creates packed array
array	**pathforallvec** –	analogous to **pathforall**
–	**pause** –	lets other processes run
x y	**pointinpath** boolean	tests whether point is in path
outcanvas incanvas outname inname detailpoint?	**postcrossings** –	generates events
string index integer	**putcard32** –	inserts 32 into string at offset
colormapsegment int color	**putcolor** –	puts color in colormapsegment
string1 string2	**putenv** –	alters value of *string1*
string	**readcanvas** canvas	reads string as canvas
event	**recallevent** –	removes event from queue
event	**redistributeevent** –	continues distribution of event
object	**refcnt** fixed fixed	returns soft and hard reference counts
object	**reffinder** –	prints references to object
canvas	**reshapecanvas** –	sets canvas to be path
canvas path width	**reshapecanvas** –	reshapes X canvas
event	**revokeinterest** –	revokes interest in event
event process	**revokeinterest** –	revokes interest in event
r g b	**rgbcolor** color	returns color object with *r g b* value
string	**runprogram** –	forks UNIX process
name object	**send** –	invokes named method in object's context
proc object	**send** –	invokes procedure in object's context
event	**sendevent** –	sends event
boolean	**setautobind** –	sets autobinding

color	**setbackcolor** –	sets **erasepage**
pixel	**setbackpixel** –	sets background pixel
canvas	**setcanvas** –	sets current canvas
color	**setcolor** –	sets current color
boolean	**setcompatinputdist** –	sets event synchronization mode
x y	**setcursorlocation** –	sets cursor location to *x y*
process	**seteventlogger** –	specifies process as event logger
any integer	**setfileinputtoken** –	adds object to tokenlist
any integer file	**setfileinputtoken** –	adds object to tokenlist
num	**setfontmem** –	sets size of font memory cache
boolean	**setkeyboardtranslation** –	tests whether translation is on
path	**setpath** –	sets path to *path*
integer	**setpixel** –	sets pixel to map index
integer	**setplanemask** –	sets planemask to integer
boolean	**setprintermatch** –	sets **printermatch** flag
num	**setrasteropcode** –	sets rasterop combination function
graphicsstate	**setstate** –	sets graphics state
–	**shutdownserver** –	aborts the NeWS server
any	**soft** boolean	tests whether argument is soft reference
any	**soften** any	returns reference as soft reference
–	**startkeyboardandmouse** –	initiates server processing
process	**suspendprocess** –	suspends *process*
num	**tagprint** –	puts *num* on output stream
any	**truetype** name	identifies true type of object
object	**typedprint** –	puts *object* on output stream
–	**unblockinputqueue** –	releases input queue block
dictionary key	**undef** –	undefines *key* from *dictionary*
–	**vmstatus** avail used size	returns status of memory usage
process	**waitprocess** value	waits until completion of process
file *or* string	**writecanvas** –	writes canvas to *file*
file object	**writeobject** –	writes object to file
file *or* string	**writescreen** –	writes screen to *file*

A.2. NeWS Operators, by Functionality

The following operators are sorted according to functionality.

Canvas Operators

width height bits/sample matrix proc	**buildimage** canvas	constructs canvas object
–	**canvasesunderpath** array	returns canvases under path
x y *or* null	**canvasesunderpoint** array	returns canvases under point
canvas	**canvastobottom** –	moves to bottom of sibling list
canvas	**canvastotop** –	moves to top of sibling list
–	**clipcanvas** –	clips to canvas boundary
–	**clipcanvaspath** –	sets current path to clip
string	**createdevice** boolean, canvas *or* env	creates canvas or environment object
canvas	**createoverlay** overlaycanvas	creates overlay canvas
–	**currentcanvas** canvas	returns current canvas

–	**eoclipcanvas** –	clips to current canvas
canvas	**eoreshapecanvas** –	reshapes canvas
file *or* string	**eowritecanvas** –	writes canvas to file
file *or* string	**eowritescreen** –	writes screen to file
canvas	**getcanvaslocation** x y	returns canvas location
–	**getcanvasshape** path	returns path object of canvas shape
–	**globalroot** canvas	gets the global root canvas
canvas	**imagecanvas** –	maps *canvas* to current canvas
boolean canvas	**imagemaskcanvas** –	analogous to **imagemask**
canvas x y	**insertcanvasabove** –	inserts above current canvas
canvas x y	**insertcanvasbelow** –	inserts below current canvas
x y	**movecanvas** –	moves canvas to *x y*
x y canvas	**movecanvas** –	moves canvas to *x y*
pcanvas	**newcanvas** ncanvas	creates new canvas
pcanvas visual cmap	**newcanvas** ncanvas	creates new canvas
string	**readcanvas** canvas	reads *string* as canvas
canvas	**reshapecanvas** –	sets *canvas* to be path
canvas path width	**reshapecanvas** –	reshapes X *canvas*
canvas	**setcanvas** –	sets current canvas
file *or* string	**writecanvas** –	writes canvas to *file*
file *or* string	**writescreen** –	writes screen to *file*

Event Operators

–	**awaitevent** event	blocks for event
null *or* num	**blockinputqueue** –	blocks input events
–	**countinputqueue** num	returns count of input queue
–	**createevent** event	creates event
event	**expressinterest** –	enables reception of events
event process	**expressinterest** –	enables reception of events
–	**getcompatinputdist** boolean	gets event synchronization mode
–	**geteventlogger** process	gets event logger process
–	**lasteventkeystate** array	returns **KeyState**
–	**lasteventtime** num	returns **TimeStamp**
–	**lasteventx** num	returns *x* coordinatate of event
–	**lasteventy** num	returns *y* coordinatate of event
outcanvas incanvas outname inname detailpoint?	**postcrossings** –	generates events
event	**recallevent** –	removes event from queue
event	**redistributeevent** –	continues distribution of event
event	**revokeinterest** –	revokes interest in event
event process	**revokeinterest** –	revokes interest in event
event	**sendevent** –	sends event
boolean	**setcompatinputdist** –	sets event synchronization mode
process	**seteventlogger** –	specifies process as event logger
–	**unblockinputqueue** –	releases input queue block

Mathematical Operators

num	**arccos** num	computes arc cosine
num	**arcsin** num	computes arc sine
a b	**max** c	leaves max on stack
a b	**min** c	leaves min on stack
–	**random** num	returns random value

Process Operators

–	**breakpoint** –	suspends current process
–	**clearsendcontexts** –	removes history of send contexts
process	**continueprocess** –	restarts suspended process
–	**createmonitor** monitor	creates monitor object
–	**currentprocess** process	returns current process
–	**currentshared** boolean	tests whether allocation status is enabled
any error name	**defaulterroraction** –	produces **$error** dictionary for process
proc	**fork** process	creates new process
–	**getprocesses** array	returns array of process groups
process *or* null	**getprocessgroup** array	returns array of processes
process	**killprocess** –	kills *process*
process	**killprocessgroup** –	kills process group
monitor procedure	**monitor** –	executes procedure with locked monitor
monitor	**monitorlocked** boolean	checks state of monitor
–	**newprocessgroup** –	creates new process group
–	**pause** –	lets other processes run
string	**runprogram** –	forks UNIX process
process	**suspendprocess** –	suspends *process*
process	**waitprocess** value	waits until completion of process

Path Operators

dx dy	**copyarea** –	copies path to *dx, dy*
–	**currentpath** path	returns current path
–	**damagepath** –	sets path to damage path
–	**emptypath** boolean	tests current path
dx dy	**eocopyarea** –	copies area to *dx, dy*
–	**eoextenddamage** –	extends damage path
–	**eoextenddamageall** –	extends damage path
–	**extenddamage** –	extends damage path
–	**extenddamageall** –	extends damage path
x y	**pointinpath** boolean	tests whether point is in path
path	**setpath** –	sets path to *path*

File Operators

listenfile	**acceptconnection** file	listens for connection
file	**countfileinputtoken** integer	returns associated usertokens
string1 string2	**file** file	creates file object
integer	**getfileinputtoken** any	returns input token from currentfile
integer file	**getfileinputtoken** any	returns input token from file
file	**getsocketlocaladdress** string	returns address of *file*
file	**getsocketpeername** string	returns name of host connected
any integer	**setfileinputtoken** –	adds object to tokenlist
any integer file	**setfileinputtoken** –	adds object to tokenlist
num	**tagprint** –	puts *num* on output stream
num file	**tagprint** –	puts *num* on output stream
object	**typedprint** –	puts *object* on output stream
object file	**typedprint** –	puts *object* on output stream
file object	**writeobject** –	writes object to file

Color Operators

color	**contrastswithcurrent** boolean	compares color with current color
visual	**createcolormap** cmap	returns new colormap for visual
cmap color	**createcolorsegment** cmapseg	returns new colorsegment
cmap C P	**createcolorsegment** cmapsegs	returns colorsegments
–	**currentbackcolor** color	gets color painted by **erasepage**
–	**currentbackpixel** integer	returns background pixel
–	**currentcolor** color	returns current color
–	**currentpixel** integer	returns index of current pixel
–	**currentplanemask** integer	returns current planemask
cmapseg integer	**getcolor** color	returns color from colormapsegment
h s b	**hsbcolor** color	returns color matching *h s b*
cmapentry int color	**putcolor** –	puts color in colormapentry
r g b	**rgbcolor** color	returns color object with *r g b* value
color	**setbackcolor** –	sets color painted by **erasepage**
pixel	**setbackpixel** –	sets background pixel
color	**setcolor** –	sets current color
colorobject *or* integer	**setpixel** –	sets pixel to map index
integer	**setplanemask** –	sets planemask to integer

Keyboard and Mouse Operators

–	**currentcursorlocation** x y	returns mouse coordinates
–	**getkeyboardtranslation** boolean	returns mode of translation
–	**getmousetranslation** boolean	tests whether events are translated
–	**keyboardtype** num	returns type of keyboard
boolean	**setkeyboardtranslation** –	tests whether translation is on
–	**startkeyboardandmouse** –	initiates server processing

Cursor Operators

–	**currentcursorlocation** x y	returns mouse coordinates
curosrchar maskchar font	**newcursor** cursor	creates cursor
cursorchar maskchar cursorfont maskfont	**newcursor** cursor	creates cursor
x y	**setcursorlocation** –	sets cursor location to *x y*

Font Operators

–	**currentfontmem** num	returns size of font memory cache
font array	**encodefont** font	duplicates font using new encoding
font name	**encodefont** font	encodes font
string	**findfilefont** font	reads font family file, returns font
font	**fontascent** number	returns font ascent
font	**fontdescent** number	returns font descent
font	**fontheight** number	returns font height
num	**setfontmem** –	sets size of font memory cache

Miscellaneous Operators

boolean errorname	**assert** –	generates an error
–	**beep** –	generates audible signal
–	**currentautobind** boolean	tests whether autobinding is on
–	**currentprintermatch** boolean	returns **printermatch** value
–	**currentrasteropcode** num	rasterop combination function
–	**currentstate** state	returns **graphicsstate** object
–	**currenttime** num	returns current time value
string index	**getcard32** integer	returns bits from offset
string1	**getenv** string2	gets value of *string1* in server
any	**harden** any	returns reference as hard reference
–	**localhostname** string	returns network hostname
–	**localhostnamearray** array	returns network hostname and aliases
file	**objectdump** –	writes summary of created objects
objects n	**packedarray** packedarray	creates packed array
array	**pathforallvec** –	analogous to **pathforall**
string index integer	**putcard32** –	inserts bits into string at offset
string1 string2	**putenv** –	alters value of *string1*
object	**refcnt** fixed fixed	returns soft and hard reference counts
object	**reffinder** –	prints references to object
name object	**send** –	invokes named method in object's context
proc object	**send** –	invokes procedure in object's context
boolean	**setautobind** –	sets autobinding
boolean	**setpacking** –	sets packing mode
boolean	**setprintermatch** –	sets **printermatch** flag
num	**setrasteropcode** –	sets rasterop combination function

graphicsstate	**setstate** –	sets graphics state
–	**shutdownserver** –	aborts the NeWS server
any	**soft** boolean	tests whether argument is soft reference
any	**soften** any	returns reference as soft reference
any	**truetype** name	identifies true type of object
dictionary key	**undef** –	undefines *key* from *dictionary*
–	**vmstatus** avail used size	returns status of memory usage

B

Byte Stream Format

The information in this chapter is only of interest to programmers implementing the NeWS protocol. Most C programmers should use CPS, which deals with all of the protocol issues transparently; see Chapter 6, "C Client Interface," for a complete discussion of the CPS facility.

The communication path between the server and a client is a byte stream: the client sends POSTSCRIPT language code to the server, and the server sends application-specific data back to the client. The basic encoding is simply a stream of ASCII characters, similar to the data stream sent to printers that understand the POSTSCRIPT language. The server also recognizes a compressed binary encoding that may be freely intermixed with the ASCII encoding. Each syntactic entity that is sent across the communication channel in this binary encoding is known as a *compressed token.* A compressed token is composed of digits, just as a POSTSCRIPT token is composed of ASCII characters. The two encodings are differentiated according to the top bit of the eight-bit bytes in the stream. If the top bit is 0, the byte is an ASCII character. If it is 1, the byte is a compressed token. This differentiation is not applied within string constants or within the parameter bytes of a compressed token.

This chapter explains the various types of compressed tokens, describes the system and user token lists, and gives a brief example of byte stream encoding. The last section of the chapter explains the concept of *tagged packets* and describes the NeWS operators that allow you to send tagged packets of data from the server to the client.

B.1. Encoding For Compressed Tokens

Each compressed token has a code in its first byte; the code is a single byte with the top bit set. Parameter bytes may follow the code byte. Parameters may also be encoded in the least significant bits of the code byte. The parameters are part of the token's description. After the code byte and any parameter bytes, there may be bytes that describe the token's value, such as an encoded integer or string.

For convenience, the various types of compressed tokens are referred to symbolically (that is, with names). The mapping between these names and their numeric values, in octal format, is given in the table below. The table also lists each token's *span,* which is the (decimal) number of possible values for that token's code byte.

Table B-1 *Token names and their associated values (in octal format)*

Value	*Span*	*Token Name*
0200	16	**enc_int**
0220	16	**enc_short_string**
0240	4	**enc_string**
0244	1	**enc_IEEEfloat**
0245	1	**enc_IEEEdouble**
0246	1	**enc_syscommon2**
0247	4	**enc_lusercommon**
0253	1	**enc_eusercommon**
0254	4	free (unused values)
0260	32	**enc_syscommon**
0320	32	**enc_usercommon**
0360	16	free (unused values)

Each type of compressed token is described below. Some of the tokens use values taken from token lists; token lists are described in more detail in Section B.2, "Token Lists."

enc_int

code byte: *enc_int*+(*d* <<2)+*w* where $0 \leq w \leq 3$ and $0 \leq d \leq 3$

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 token type is used to encode integers and fixed-point numbers.

enc_short_string

code byte: *enc_short_string*+*w* where $0 \leq w \leq 15$

The next *w* bytes are taken as a string.

enc_string

code byte: *enc_string*+*w* where $0 \leq w \leq 3$

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. Note that there may be an implementation limit on the maximum size of a string.

enc_IEEEfloat

code byte: *enc_IEEEfloat*

The next four bytes, high order to low order, form an IEEE format floating-point number.

enc_IEEEdouble

code byte: *enc_IEEEdouble*

The next eight bytes, high order to low order, form an IEEE double precision floating-point number.

enc_syscommon

code byte: *enc_syscommon*+*k* where $0 \leq k \leq 31$

Inside the X11/NeWS server is a list of POSTSCRIPT language objects known as the system token list; the names in the list are objects that represent POSTSCRIPT language operators and NeWS operator extensions. The **enc_syscommon** token causes the *k*th entry in the list to be inserted in the input stream. This list is a

constant for all instances of POSTSCRIPT language code; the contents of the system token list are fixed. This token type allows the most common POSTSCRIPT language operators to be encoded as a single byte. (The most common operators are listed in the first 32 list entries.)

enc_syscommon2

code byte: *enc_syscommon2* parameter byte: *k* where $0 \le k \le 255$

This token type is essentially identical to **enc_syscommon** except that the index into the system token list is $k+32$. Thus, the 33rd entry in the list is represented with a *k* of 0. This token type allows less common NeWS operators to be encoded as two bytes: one code byte and one parameter byte.

enc_usercommon

code byte: *enc_usercommon+k* where $0 \le k \le 31$

This token type is similar to **enc_syscommon** except that it provides user-definable tokens. Each communication channel to the server (represented by a file) has an associated user token list. The **enc_usercommon** token causes the *k*th list entry to be inserted in the input stream. The list is dynamic; it is the responsibility of the client program to load objects into this list. The NeWS operator **setfileinputtoken** associates an object with a list position for an input channel (see Chapter 10, "NeWS Operator Extensions"). C clients can also use CPS utilities for manipulating the user token list (see Chapter 6, "C Client Interface").

The **enc_usercommon** token type allows the most commonly-used user tokens to be encoded as one byte. The **enc_usercommon** tokens occupy slots 0 through 31 in the user token list.

enc_lusercommon

code byte: *enc_lusercommon+j* parameter byte: *k* where $0 \le j \le 3$ and $0 \le k \le 255$

This token type is essentially identical to **enc_usercommon** except that the index is $(j<<8)+(k+32)$. This token type allows the less common user tokens to be encoded as two bytes: one code byte and one parameter byte. The **enc_lusercommon** tokens occupy list positions 32 through 1055.

enc_eusercommon

code byte: *enc_eusercommon* parameter bytes: *j*, *k* where $0 \le j \le 255$ and $0 \le k \le 255$

This token type is essentially identical to **enc_usercommon** except that the index into the user token list is $(j<<8)+(k+32+1024)$. This encoding simply expands the user token list to a total of 65,536 entries. The **enc_eusercommon** tokens occupy list positions 1056 through 65,535. These tokens should be used for the least common user tokens, since this encoding requires three bytes (one code byte and two parameter bytes).

B.2. Token Lists

The **enc_*common*** tokens all interpolate values from token lists. The appearance of one of these tokens causes the appropriate entry of a token list to be used as the value of the token. These tokens are typically part of a POSTSCRIPT language stream that is to be executed.

The entries in the token lists can be any kind of POSTSCRIPT language object; they are usually either executable name objects or operator objects. If the object is a name, its value is looked up before being executed, just as an ASCII encoded name is looked up. If the object is an operator, it is executed directly, or it is

inserted into an executable array if one is currently being parsed. This behavior improves performance, but it also binds the interpretation of the token to the value of the object at the time that the object is loaded into the token list.

For example, if the executable name **moveto** is loaded into a list, then whenever that token is encountered **moveto** is looked up and executed. However, if the value of **moveto** is loaded into the list, then whenever that token is encountered the interpretation of **moveto** *at the time it was loaded* is used.

System Token List

The system token list (used by the **enc_syscommon*** tokens) is located in `$OPENWINHOME/etc/NeWS/systoklst.ps`. This file contains a list of common POSTSCRIPT language operators and NeWS operator extensions. The first 32 entries in the list are represented in the input stream by tokens of type **enc_syscommon**, and the remaining entries are represented in the input stream by tokens of type **enc_syscommon2**.

Each object in the list has an index equal to its position in the list. For example, the first object has an index value of 0. Thus, the encoding for the first list entry, in octal format, is 0260 + 0 = 0260.

The contents of the system token list are fixed.

User Token List

Each server file object has a user token list associated with it. Because each connection to the server is represented by a file, each connection has a user token list available. The client can add tokens to the user token list with the **setfileinputtoken** operator or with CPS utilities.

The user token list has a maximum of 65,536 entries. The first 32 entries are represented in the input stream by the **enc_usercommon** token type, the next 1024 entries by the **enc_lusercommon** token type, and the last 64,480 entries by the **enc_eusercommon** token type.

B.3. Encoding Example

Assume that you want to encode the following fragment of POSTSCRIPT language code:

```
10 300 moveto
(Hello world) show
```

This code fragment can be encoded simply as the following ASCII text string:

```
"10 300 moveto\n(Hello world) show "
```

The ASCII encoding produces 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 this code fragment is sent in compressed binary format, the message would be the 19 bytes shown in the following table:

Table B-2 *Bytes for binary encoding example (given in octal format)*

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
0274	(0260+014) **moveto** is in slot 12 of the system token list
0233	(0220+11) Start of an 11-character string
0110	'H'
0145	'e'
...	
0144	'd'
0304	(0260+024) **show** is in slot 20 of the system token list

B.4. Sending Tagged Replies from Server to Client

The client can download NeWS code that sends replies back from the server. These replies should be sent as *tagged packets* of information. Each tagged packet consists of an integer tag followed by typed data. A packet's tag separates its data from the data of another packet; the tags allow the client to identify packets that are returned asynchronously.

The server provides two operators, **tagprint** and **typedprint**, that allow the client to send tagged replies back from the server. These two operators are described below.

n **tagprint** –
n file **tagprint** –

Prints the integer *n* (where $-2^{15} \leq n < 2^{15}$) encoded as a tag on the specified output *file*; if no *file* is specified, prints *n* on the current output stream. The tag can then be read from the client's input connection file.

object **typedprint** –
object file **typedprint** –

Prints *object* in an encoded form on the specified output file; if no *file* is specified, prints *object* on the current output stream. The object can then be read from the client's input connection file.

The client's downloaded code should call **typedprint** immediately after **tagprint**, without pausing, to ensure that the packet is not mixed with another packet. Monitors may be required to ensure proper synchronization. The **tagprint** and **typedprint** operators send the data in compressed form.

See Chapter 6, "C Client Interface," for information about using **tagprint** and **typedprint** in the context of the CPS facility.

C

The Extended Input System

This appendix contains information on the *Lite* user interface. This interface, previously available under NeWS 1.1, continues to be supported but will no longer be enhanced.

C.1. Building on NeWS Input Facilities

The *Extended Input System* (EIS) described in this appendix is implemented entirely in the POSTSCRIPT language on top of the basic facilities provided by the primitives in the NeWS server. It aims to support a sophisticated interface of at least the complexity of SunView or the Mac, and to provide at least one such interface as an existence proof. It also is aimed at separating independent issues in the implementation of interfaces. For example, it should be possible to provide alternatives in each of the following three categories without dependencies between categories and without requiring any change to client code:

- different input devices (1– and 3–button mice, or keyboards with different collections of function keys);
- alternative styles of input-focus, such as follow-cursor or click-to-type;
- alternative styles of selection, such as point-and-extend or wipe-through.

The EIS is sufficiently flexible that it should be possible to support a keyboard-only input system.

This chapter has several independent sections, corresponding to some of the modules of the EIS. It begins with a description of a particular user interface, implemented by the file `liteUI.ps`, which is a suggestive subset of the SunView interface. It includes a description of the requirements and facilities for a client to handle keyboard input and selections in that world.

A good deal of the processing in the EIS is carried on in a single process called "the global input handler." Some of it, however, must be done on a per-client basis; facilities are provided which are active in the client's lightweight process in the server. For example, recognizing events that indicate a change of input focus and distributing keystrokes to that focus are done in the global input handler. But recognizing user actions that indicate a selection is to be made must be done for each client, since some clients will not make selections at all, but will apply other interpretations to the same user actions.

C.2. The LiteUI Interface

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

Primary, Secondary, and Shelf selections are provided; [Copy] and [Paste][7] work with all of them in the standard fashion. Selections are made when the **ViewPoint** mouse button goes down, and are always in character units. Keyboard focus may be controlled either by cursor motion into and out of windows, or by clicking a mouse button to reset the focus. In the latter mode, the **ViewPoint** (this is in **UserProfile**, and is set to **LeftMouseButton** by default) button sets both the focus and the Primary selection at the indicated position; the **ViewAdjust** (**MiddleMouseButton** by default) button restores the focus to a window, at its previous position, and without affecting the Primary selection.

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

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

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

[7] These are the names of the keys on the keyboard in SunOS 4.0; however, internally the EIS refers to them as "Put" and "Get" operations.

C.3. Keyboard Input

Keyboard Input: Simple ASCII Characters

Four procedures provide access to increasingly sophisticated levels of keyboard input. The most straightforward client merely wants to get characters from the keyboard. To do this call **addkbdinterests**, passing the client canvas as an argument; then enter a loop, doing an **awaitevent** and processing the returned event.

addkbdinterests

canvas **addkbdinterests** [events]
declares the client's canvas to be a candidate for the input focus. It also creates and expresses interest in the following three kinds of events, and returns an array of the three corresponding interest-events:

ASCII Typing

The first interest has **ascii_keymap** for its **Name**, and **/DownTransition** for its **Action. ascii_keymap** is a dictionary provided by EIS for expressing interest in ASCII characters; it includes the translation from the user's keyboard to the ASCII character codes where that is necessary. Events which match this interest will have ASCII characters in their **Names**, and **/DownTransition** in their actions. The client can choose to see up-events too, by storing **null** into the **Action** field of this interest.

Inserting Text

The second interest has the name **/InsertValue** and a **null Action**. This will match events whose **Name** is the keyword **/InsertValue**, and whose **Action** is a string which is to be treated as though it had been typed by the user. Such events will be generated if some process is pasting selections to this window, or if function-key strings have also been requested (see below).

Input Focus

The third interest has the array [**/AcceptFocus /LoseFocus /RestoreFocus**] in its **Name.** Events matching this interest inform the client that it now has, or has lost, the input focus. These events are informational only; they do not affect the distribution of keyboard events. They are intended for clients which provide some feedback, such as a modified namestripe or a blinking caret, when they have the input focus. Clients are always free to ignore them.

Revoking Interest in Keyboard Events

A process that is about to exit, or that will continue to exist, but wants no more keyboard input, may revoke its interest in keyboard input by passing the array returned from **addkbdinterests**, along with the client canvas, to **revokekbdinterests**:

revokekbdinterests

[events] canvas **revokekbdinterests** –
Undoes all the effects of **addkbdinterests**.

Keyboard Input: Function Keys

By default, clients do not receive any events associated with function keys. A client can choose to receive function-key events, either in the form of a keyword naming the key that went down, or as a string of the form `"ESC[`*nnn*`z"` (the ASCII-standard escape sequence for such keys).

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

addfunctionstringsinterest

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

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

addfunctionnamesinterest

canvas **addfunctionnamesinterest** event
creates an interest in the function keys, expresses interest in it, and returns that event. As a result, when a function key is pressed, **awaitevent** returns an event whose **Name** is a keyword like **/FunctionL7**.

By default, both up and down transitions on the keys are noted; the client may change this by storing **/DownTransition** (or **/UpTransition**, if that is what is desired) into the **Action** field of the returned interest. **addkbdinterests** must also have been called for this procedure to have any effect.

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

Assigning Function Keys

Users may assign a procedure to be executed when a specified key goes down. See the section on **bindkey** in Chapter 11, *Extensibility through NeWS Procedure Files*.

Keyboard Input: Editing and Cursor Control

If the client is passing characters through to a shell or some similar process that will do its own translations on them, it should pass them through unmodified. But if the client is dealing with text directly, it should provide the editing and caret-motion facilities defined in the user's global profile. To assist in this, the client may ask for incoming events to be checked for a match against those keyboard actions, and converted to uniform editing-events if they do. This is done by passing the client canvas to **addeditkeysinterest**.

addeditkeysinterest

canvas **addeditkeysinterest** event
creates an interest in the key combinations that are defined for global editing and caret motion, expresses interest in it, and returns that event. As a result, the client sees events with a **Name** from the set:

{Edit,Move}{Back,Fwd}{Char,Word,Field,Line,Column}

For example, here are the event names for the various **EditBack*** combinations:

/EditBackChar Delete the character before the caret.

/EditBackWord Delete the word before the caret.

/EditBackField Move the caret back to the end of the preceding field if any exists, deleting its contents or selecting them in pending-delete mode.

/EditBackLine Delete from the caret back to the beginning of the current line.

/EditBackColumn Delete all characters between the caret and the nearest boundary in the line above; if the previous line ends to the left of the caret, delete back through the preceding end-of-line.

Substituting *Fwd* for *Back* indicates that the deletion or motion (see the next paragraph) extends *after* rather than before the caret. **/EditFwdLine** deletes through the next end-of-line.

Substituting *Move* for *Edit* indicates the caret is moved to the far end of the span that would be deleted by an **Edit**, but the characters are not deleted.

Again, no separate procedure is provided to revoke this interest, since the **revokeinterest** primitive does exactly what is needed.

C.4. Selections

Clients that will make selections and pass information about them to other processes declare this interest by calling **addselectioninterests**. Thereafter, EIS code will process user inputs according to the current selection policy. Occasionally, it will pass a higher-level event through to the client, when some client action is required in response. The exact interface by which a user indicates a selection is not the client's responsibility; the client must simply be prepared to handle higher-level events. Clients will also occasionally see events with a **Name** of **/Ignore**; these are events which were delivered to the client process, but handled entirely by EIS code before the event was made available to the client. The **/Ignore** event is left behind in this situation so that client code can depend on an event being on the stack when it gets control after **awaitevent** returns.

Selection Data Structures

There is no separate "selection service" in EIS; some selection processing takes place in the global input handler, and the rest in client processes. There is a global repository of data about selections, however, and there are some standard formats for the information stored in that repository and communicated between selection clients.

There are two broad styles of dealing with selections: the *communication model* and the *buffer model*. In the former, the selection holder says *"Here is my phone number, call me for answers about the selection I hold."* In the buffer model, the selection holder puts all the information about its selection into the selection-dict itself.

The selection most users are familiar with is the primary selection indicating the text they have selected in a terminal emulator window. However, there are other kinds of selection. A selection is named by its *rank*; in *liteUI*, the ranks are **/PrimarySelection**, **/SecondarySelection**, and **/ShelfSelection**.[8] For each rank, there is a dictionary containing the information known to the system about that selection. Such a dictionary will be called a *selection-dict* henceforth. It will have at least the following three keys defined:

---

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

Table C-1 *Selection-Dict Keys*

Key	Type	Semantics
SelectionHolder	process	Which process made the selection
Canvas	The canvas	the canvas in which the selection was made.
SelectionResponder	null or process	What process will answer requests concerning this selection.

If **SelectionResponder** is defined to **null**, then the selection holder is using the buffer model, and information about the selection will be stored in other keys defined in the dictionary. setting out all available information about that selection. A few such keys have been defined because they are expected to be generally useful. These are listed in the table below. Others may be provided by clients as convenient — there is no limit on what consenting clients may say to each other kbout a selection.

Table C-2 *System-defined Selection Attributes*

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

Finally, communications between clients about selections (that is, requests and their responses) are formatted as another dictionary, hereafter called a *request-dict.* When submitted by the requester, the dictionary will have a key naming each attribute for which the requestor wants a value. It may also contain commands the selection holder should execute, such as **ReplaceContents**. When received by a selection holder, a request-dict will contain the keys defined by the requester, plus the following two:

Table C-3 *Request-dict Entries*

Key	Type	Semantics
Rank	rank	the rank of selection which this request concerns
SelectionRequester	process	the process which is sending the request

NOTE *When a NeWS client has the selection and the user makes a new selection in an XView client, the XView client generates a* **SUN_SELN_YIELD** *key in the request-dict. It is generally safe for NeWS clients to ignore such requests.*

The use of these various structures is detailed under the relevant event descriptions below.

Selection Procedures

This section lists the library procedures provided for clients to deal with selections.

addselectioninterests

canvas **addselectioninterests** [events]
creates and expresses interest in two classes of events, returning an array of the two interests.

The first interest matches events with names in the following list:

Table C-4 *High-Level Selection-Related Events*

/InsertValue
/SetSelectionAt
/ExtendSelectionTo
/DeSelect
/ShelveSelection
/SelectionRequest

The response required from the client to each of these events is detailed below under *Selection Events*. (Some clients may safely omit handlers for the last two; see the detailed description).

The second interest matches events which are uninteresting to the client. It arranges for EIS processing to be done by library code before the client ever sees the event.

clearselection

rank **clearselection** –
sets the indicated selection to **null**; this allows a selection holder to indicate the selection no longer exists.

selectionrequest

request-dict rank **selectionrequest** request-dict

makes a request (contained in *request-dict*) concerning the selection of the specified *rank*. The format of a *request-dict* is described above, in Table C-3, *Request-dict Entries*. The **SelectionRequester** and **Rank** entries will be filled in by **selectionrequest**.

If the **SelectionResponder** in *rank*'s selection-dict is null , then the selection holder is employing the buffer model. The **selectionrequest** procedure itself fills in the request-dict using information which the selection holder put in the selection-dict. But if the **SelectionResponder** in *rank*'s selection-dict is not null, then the selection holder is employing the communication model, and **selection-request** has to do a lot more work. It sends the request-dict to the **Selection-Responder** process in a **/SelectionRequest** event, and forks a process that waits for a reply. The **SelectionResponder** process is supposed to fill in the request-dict with whatever values the requester asked for, then hand back the same dictionary using **selectionresponse**; this is explained in greater detail in the description of **/SelectionRequest** below. If the **SelectionResponder** process does not respond within a certain amount of time, **selectionrequest** will return **null**.

In either case, if the indicated selection does not exist, **selectionrequest** will return **null**. Also, some keys in the request may not have an answer available. In this case they will be set to **/UnknownRequest** in the response.

selectionresponse

event **selectionresponse** –

is used by a selection holder using the request-model to return a response when it receives a selection request event. The *event* given should be the same **/SelectionRequest** event which the selection holder has just processed. (**/SelectionRequest** events are described below under *Selection Events*.) **selectionresponse** transforms *event* into a **/SelectionResponse** event and returns it to the requester.

setselection

selection-dict rank **setselection** –

is used by a process to declare itself the holder of a selection. *selection-dict* is a dictionary containing either a definition of **SelectionResponder**, or of keys which provide data about the selection itself, as described above in Table C-1, *Selection Data Structures*. *Rank* indicates which selection is being set. If another process currently holds that selection, it will be told to deselect.

getselection

rank **getselection** selection-dict

retrieves the information currently known to the system about the indicated selection. This procedure is likely to be more useful to the implementor of a package like *liteUI* than to window clients.

Selection Events

As mentioned above, clients may expect to receive six different kinds of events concerning the selection. Of these, the **/InsertValue** event has already been described under *Keyboard Input*; its usage in the selection context is exactly the same as for function strings. The remaining five events and the appropriate responses to them are presented below.

Each event is described in the following format:

EventType *short description of the event's semantics*

Name:
keyword that identifies the event

Action:
description of the contents of the event's **Action** *field*

Response:
description of what the client should do when it receives such an event

/SetSelectionAt

Informs the client the user has just made a selection in its canvas.

Name:
/SetSelectionAt

Action:

dict [	**Rank**	**/PrimarySelection | /SecondarySelection**
	X	*number*
	Y	*number*
	PendingDelete	**true | false**
	Preview	**true | false**
	Size	*number*
]		

NOTE *LiteUI provides constant values for three fields:* **PendingDelete = false, Preview = false,** *and* **Size** = *1.*

Response:

Make a selection of the indicated **Rank** with the following parameters:

Key	*Value*
X and **Y**	indicate a position (it will be in the current canvas' coordinate system).
Size	indicates the unit to be selected; for example, in text: 0 means a null selection at the nearest character boundary, 1 corresponds to a character, and larger values indicate larger units (words, lines, etc.) whose definition is at the discretion of the client
PendingDelete	indicates whether that mode should be used (if supported by the client)
Preview	indicates whether the selection is only for feedback to the user; a selection shouldn't actually be set until a selection event is received with **Preview false**

In client POSTSCRIPT langauge code, some private processing will generally be required. For instance, the given position will have to be resolved to a character in a text window, and appropriate feedback displayed on the screen. Then the client should build a selection-dict describing the selection just made, and pass it to **setselection**, along with the rank it received in the **/SetSelectionAt** event:

```
selection-dict rank setselection
```

'selection-dict' should contain either a non-null definition of **Selection-Responder**, or it should define keys which actually provide information about the selection (**ContentsAscii** at a minimum). In the former case, the holder is following the *communication model* of selection, and must be prepared to receive and respond to **/SelectionRequest** events as long as it holds the selection. In the latter case, the holder is following the *buffer model* of selection; requests will be answered automatically by the global input handler.

'selection-dict' will have keys added to it, so it should be created with room for at least *five* more entries beyond those defined by the client.

/ExtendSelectionTo

Tells the client the user has just adjusted the bounds of a selection in its canvas.

Name:
  /ExtendSelectionTo

Action:

dict [	Rank	**/PrimarySelection | /SecondarySelection**
	X	*number*
	Y	*number*
	PendingDelete	**true | false**
	Preview	**true | false**
	Size	*number*
]		

Response:

The dictionary in the **Action** field is the same as the **Action** of a **/SetSelectionAt** event, and the client response is very much the same. The distinction is that this event indicates a modification of an existing selection, where **/SetSelectionAt** indicates a new one.

The client should adjust the nearest end of the current selection of the indicated **Rank** to include the indicated position. If **Size** indicates growth, extend both ends as necessary to get them at a boundary of the indicated size. (For example, if **Size** has changed from 1 to 2, a text window might grow both ends of the selection to ensure that they fall at word boundaries.) Adjust the **PendingDelete** mode or ignore it as the window is editable or not.

If there was no selection of the indicated rank, pretend there was an empty one at the indicated position.

In client POSTSCRIPT language code, after doing any private processing required, processing is exactly the same as for **/SetSelectionAt**.

/DeSelect

Informs the client that it no longer holds the indicated selection.

Name:

/DeSelect

Action:

rank

Response:

Undo a selection of the given rank in this window. *Do not* call **clearselection**; the global selection information has already been updated.

/ShelveSelection

Tells the client to set the shelf selection to be the same as a selection which the client currently holds.

Name:

/ShelveSelection

Action:

rank

Response:

Buffer-model clients (those that did not define **SelectionResponder** when they set the selection) will not receive **/ShelveSelection** events; the service will copy their selection to the shelf for them. Others should set the **ShelfSelection** to be the same as the selection whose *rank* is in **Action**, using **setselection** as above.

NOTE *Be careful of the difference between the* **ShelveSelection** *and* **ShelfSelection***; the former is a selection event, and the latter is one of the selection ranks along with* **/PrimarySelection** *and* **/SecondarySelection**.

/SelectionRequest

The client is requested to provide information about a selection it holds.

Name:
/SelectionRequest

Action:
request-dict

Response:
Buffer-model clients (those that did not define **SelectionResponder** when they set the selection) will not receive **SelectionRequest** events; the service will answer the request for them.

The client should enumerate the request-dict, responding to the various requests by defining their values (as for **ContentsAscii**), or performing the requested operation (as for **/ReplaceContents**, whose value will be the replacement value). The resultant dict should be left as the **Action** of the event, which should then be passed as the argument to the procedure **selectionresponse**.

NOTE *There is no restriction on what requests may be contained in a selection request; this is left to negotiation between the requester and the selection holder. A holder may reject any request, by defining its value to be* **/UnknownRequest**.

It may be noted that there is no mechanism described here for getting a selection's contents from someplace else. In *liteUI*, user actions that precipitate such a transfer are recognized and processed in the global input handler, which then performs the selection request, and sends an **/InsertValue** event to the receiving process. The selection procedures described above provide an interface for instigating such transfers independent of user actions.

C.5. Input Focus

The input focus (where standard keyboard events are directed) is maintained by the global input-handler process, according to the current focus policy. A client becomes eligible to be the input focus by calling **addkbdinterests** (described above under *Selection Procedures*). At some later time, some user action will indicate that the client should become the focus. The client will receive an event indicating this has happened (its **Name** will be **/AcceptFocus** or **/RestoreFocus**, and its **Action** is described in the table below). Thereafter, the client will receive events whose **Names** are ASCII character codes. Loss of the keyboard focus will be indicated by the delivery of an event with **Name /LoseFocus**.

Table C-5 *Input Focus*

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

Table C-5 *Input Focus— Continued*

Name	*Action*	*Explanation*
	5	The canvas directly or indirectly contains the pointer and used to be a descendant of the focus. The new canvas is not equivalent to this canvas nor is the new canvas an ancestor or descendant of this canvas.
	6	The focus used to be **PointerRoot**.
	7	The focus used to be **None**.

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

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

setinputfocus

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

currentinputfocus

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

hasfocus

process **hasfocus** bool
Returns **true** or **false** as the indicated process is or is not currently the input focus.

setfocusmode

keyword **setfocusmode** –
The global focus policy is reset to the policy named by the argument. Currently-supported focus policies are identified by:

/ClickFocus As long as no function keys are down clicking the Select button will set both the focus and the primary selection in a window. Clicking Adjust will restore the focus at its last position in this window, without making any selection.

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

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

D

Omissions and Implementation Limits

D.1. Operator Omissions and Limitations

The following primitives are defined in the *PostScript Language Reference Manual.* They have not been implemented in the X11/NeWS POSTSCRIPT language interpreter because they are either printer- or environment-specific.

banddevice	**renderbands**
framedevice	**start**

The following operators are implemented, but they do not do anything. If you execute them they will consume or produce the right arguments on the operand stack, but they will have no other effects. The **showpage** operator does perform an implicit **initgraphics** operation, but it otherwise has no other effect.

copypage	**showpage**
echo	

The following operators are unimplemented:

executeonly	**resetfile**
noaccess	**reversepath**

For Folio fonts, the **charpath** operator returns the actual path outline of the string given as an argument; for other fonts, it returns the bounding rectangle of this path.

Some loop operators (such as **forall**) may add more than one level to the execution stack. This is important to know only if you are examining an execution stack dumped by a stack overflow error.

D.2. Font Dictionary Limitations

The following font dictionary keys are not supported under NeWS 2.1:

FamilyName	**Private**
FullName	**StrokeWidth**
Metrics	**Version**
Notice	**Weight**

D.3. The statusdict Dictionary

Most of the entries in the **statusdict** dictionary (described in Appendix D of the *PostScript Language Reference Manual*) are pseudo-implemented; they have reasonable values, but setting them has no effect. One exception is the job timeout. Getting and setting the job timeout will change how long a process is allowed to execute without blocking before receiving a `timeout` error.

D.4. Implementation Limits

The following table lists implementation limits of NeWS:

Table D-1 *Implementation Limits*

Quantity	*Limit*	*Explanation*
integer	$\pm 2^{31}$	Integers are represented as 32 bits. Integers are automatically converted to reals if they overflow.
real		Single-precision floating-point numbers are used. Some reals are represented using fixed point numbers with 16 bits of whole number and 16 bits of fraction; other reals are represented using single precision floating point numbers. Reals are represented as fixed point numbers if they are small enough, but the type determination operators will describe them as real.
array	65535	Number of entries in an array.
dictionary	65535	Number of key/value pairs in a dictionary.
string	65535	Number of characters in a string.
name	65535	Number of characters in a name.
file		Number of open files (includes open client communication channels). The limit is `getdtablesize()`-*n*, where *n* depends on the particular server but will be about four.
userdict	50000	Initial number of entries. Set in `init.ps`. To change, modify the value assigned to /DICTBEGINSIZE in `util.ps`.
operand stack	1500	Maximum number of operands on the stack.
dict stack		Expanded as required.
exec stack	250	Maximum function and/or compound statement nesting depth.
gsave level		Expanded as required.
path		Expanded as required.

Table D-1 *Implementation Limits— Continued*

Quantity	*Limit*	*Explanation*
VM		The server expands to use as much VM as the underlying system permits.
interpreter level		Not applicable.
save level		Expanded as required.

D.5. Other Differences with the POSTSCRIPT Language

In addition to the omissions and differences implemented above, the POSTSCRIPT language has slightly different semantics for some standard POSTSCRIPT language operators. The NeWS versions of these operators are described in Chapter 10, *NeWS Operator Extensions* along with the wholly-new NeWS operators.

Table D-2 *NeWS Versions of Various POSTSCRIPT Language Operators*

Operator	*Note*
==	Although there is no specification for their output in the *PostScript Language Reference Manual*, you may be confused because = and == print objects out in a slightly different format than the particular implementations in the LaserWriter. = identifies dictionaries as such and prints some useful fields from the various “magic dictionary” types in NeWS. == actually prints out the first few key-value pairs in dictionaries. == uses quotation marks as follows to indicate the types of objects: '*operator-type*'.
bind	**bind** is implemented in NeWS, but it is useful only when autobinding is off. Autobinding is on by default. See **currentautobind** in Chapter 9, *NeWS Operator Extensions*.
file	NeWS has the additional special name ‘`%socket`*n*’ for socket-based network connections to the server. Also, the POSTSCRIPT language operators **file** and **run** (together with all the NeWS utility procedures for file access) will look for relative path names in the directory from which the server was started, *and* in `$OPENWINHOME/etc`.

Table D-2 *NeWS Versions of Various POSTSCRIPT Language Operators— Continued*

Operator	*Note*
vmstatus	In NeWS, **vmstatus** returns *avail*, *used*, and *size*; the *PostScript Language Reference Manual* states that it should return *level*, *used*, and *maximum*.

Index

Special Characters
<errors>, 215
?get, 300
?getenv, 300
?load, 300
?put, 300
?revokeinterest, 302
?undef, 300

A

acceptconnection, 257, 270
accessing files, 305
Action, 247
addeditkeysinterest, 332
addfunctionnamesinterest, 332
addfunctionstringsinterest, 332
addkbdinterests, 331
addselectioninterests, 335
/AllEvents, 239
animated drawing procedures, 40, 262
append, 293
arccos, 257
arcsin, 257
arraycontains?, 297
arraydelete, 297
arraydelete!""
  arraydelete, 297
arrayindex, 297
arrayinsert, 298
arrayop, 298
arrayreverse, 298
arrayreverseFast, 298
arrays, 297
  size limit, 346
arraysequal?, 298
assert, 257
asynchronous replies from server, 185, 188
AutoRepeat, 245
awaitevent, 257, 85, 262, 267, 282, 331
awaiting events, 85

B

banddevice, 345
`basics.ps`, 291
beep, 258
BellPercent, 245
BellPitch, 245
bindkey, 309
BindOverride, 251
BitsPerPixel, 256
blockinputqueue, 258, 122, 257, 288
BorderWidth, 240
BottomCanvas, 36, 237
breakpoint, 258, 69, 261, 287
buildimage, 258, 49, 50, 271, 272
buildsend, 293
byte stream format, 323
  encoding example, 326
  system token list, 326
  tagprint, 327
  token encoding, 323
  typedprint, 327
  user token list, 326

C

C client interface, *see* CPS facility
cache for fonts, 231
canvas clipping operators, 33
canvas clipping path, 33
canvas damage, 13, 28
canvas damage, repairing, 29
canvas hierarchy, 10
canvas hierarchy, manipulating, 35
canvas interest lists, 96
canvas location, getting, 22
canvas operators, 8
Canvas, 247
CanvasAbove, 36, 237
CanvasBelow, 36, 238
canvases, 3, 7
  /AllEvents, 239
  ancestors, 12
  animated drawing, 40, 262
  BorderWidth, 240
  bottom child, 12
  BottomCanvas, 36, 237
  buildimage, 49, 50
  canvas clipping path, 33
  canvas hierarchy, 10
  CanvasAbove, 36, 237
  CanvasBelow, 36, 238

canvases, *continued*
- **canvastobottom**, 36, 260
- **canvastotop**, 37, 260
- **canvastype** extension, 7
- changing parenthood, 39
- changing sibling relationships, 36
- **clipcanvas**, 33
- **clipcanvaspath**, 33, 261
- clipping operators, 33
- **Color**, 239
- **Colormap**, 240
- coordinate systems, 14
- **createdevice**, 262
- **createdevicecanvas**, 55
- **createoverlay**, 41, 262
- creating canvases, 15
- current canvas, 18
- current transformation matrix, 14
- **currentcanvas**, 263
- **Cursor**, 51, 240
- cursors, 51
- damage, 13, 28
- **damagepath**, 265
- default transformation matrix, 14
- descendants, 12
- destroying, 22
- drawing in canvases, 18
- **eoclipcanvas**, 34
- **eoreshapecanvas**, 267
- **eowritecanvas**, 267
- **eowritescreen**, 267
- **EventsConsumed**, 99, 239
- **fillcanvas**, 18
- framebuffer, 15
- framebuffer canvas, 10, 15
- **framebuffer**, 308
- **getcanvaslocation**, 22
- **getcanvasshape**, 268
- getting location, 22
- **Grabbed**, 241
- **GrabToken**, 241
- **imagecanvas**, 47, 271
- **imagemaskcanvas**, 49, 272
- **imagepath**, 272
- imaging, 44, 50
- **insertcanvasabove**, 37, 272
- **insertcanvasbelow**, 37, 272
- **Interests**, 239
- manipulating the hierarchy, 35
- **Mapped**, 17, 238
- mapping, 13
- mapping to the screen, 17
- **/MatchedEvents**, 239
- **movecanvas**, 21, 274
- moving, 20
- **newcanvas**, 15, 274
- **newcursor**, 53
- **/NoEvents**, 239
- non-retained, 13
- opaque and transparent, 12
- opaqueness, 22
- operator extensions, 8
- overlay canvases, 40, 262
- **OverrideRedirect**, 240

canvases, *continued*
- **Parent**, 39, 238
- **readcanvas**, 47, 279
- reading and writing to files, 44
- **rectpath**, 16
- repairing damage, 29
- **reshapecanvas**, 16, 280
- retain threshold, 29
- retained, 13
- retained canvases, 29
- **Retained**, 238
- **RowBytes**, 241
- **rrectpath**, 29
- **SaveBehind**, 32, 239
- **setcanvas**, 18, 283
- setting coordinate system, 16
- shaping, 16
- **SharedFile**, 240
- sibling canvases, 12, 36
- **SubstructureRedirect**, 240
- top child, 12
- **TopCanvas**, 36, 237
- **TopChild**, 36, 238
- transparency, 22
- **Transparent**, 22, 238
- unrooted canvases, 14
- visibility, 13
- **VisibilityInterest**, 240
- **Visual**, 240
- **VisualList**, 240
- **writecanvas**, 45, 288
- **writescreen**, 45, 289
- **XID**, 240

canvasesunderpath, 259
canvasesunderpoint, 260, 264
canvastobottom, 260, 36, 272
canvastotop, 260, 37, 272
canvastype dictionary keys, 7, 236 *thru* 241
canvastype extension, 236
canvasesunderpoint, 260
case, 293
`cdef` command, 180, 183
- asynchronous replies, 188
- requesting no reply, 186
- synchronous replies, 186

changing and reusing events, 84
changing canvas parenthood, 39
cid utilities, 306
cidinterest, 307
cidinterestlonly, 307
/class, 176
Class, 255
classbegin, 139
classdestroy, 165
classend, 139
classes, **131**, 6
- batch send, 153
- **/class, 176**
- class initialization, 140
- class **Object**, 133
- class variables, 132
- class, definition of, 131

classes, *continued*
- `class.ps`, 292
- **classbegin, 139**
- **classdestroy, 165**
- **classend, 139**
- **/classname, 175**
- **/cleanoutclass, 165**
- creating a new class, 139
- creating a new instance, 156
- default class, 161
- **/defaultclass, 162**
- **/descendantof?, 176**
- **/destroy, 165**
- destroying classes, 165
- destroying instances, 165
- **/doit, 153**
- immediate superclass, 133
- inheritance, 134
- inheritance array, 135
- **/installmethod, 153**
- instance variables, 132
- instance, definition of, 131
- **/instanceof?, 176**
- intrinsic class, 161
- **isclass?, 175**
- **isinstance?, 175**
- **isobject?, 175**
- list of class methods, 177
- list of class operators, 177
- method compilation, **150**, 147, 149
- **/methodcompile, 152**
- methods, 132
- multiple inheritance, 134, 166 *thru* 174
- **/name, 174**
- **/new, 157**
- **/newdefault, 161**
- **/newinit, 158**, 157
- **/newmagic, 159**
- **/newobject, 157**, 157
- **Object**, 292
- object, definition of, 131
- **/obsolete, 166**
- obsolete classes, 165
- obsolete instances, 165
- overriding class variables, 162
- **promote, 163**
- **promoted?, 164**
- promoting class variables, 163
- **redef, 140**
- **self**, 145 *thru* 150
- **send**, 140 *thru* 145
- send context, 140
- **/sendtopmost, 177**
- **SetLocalDicts, 154**
- **/setname, 174**
- subclass, 133
- **/subclasses, 176**
- **/SubClassResponsibility, 162**
- **super**, 145 *thru* 150
- superclass, 133
- **/superclasses, 175**
- the class tree, 133
- **/topmostdescendant, 176**
- **/topmostinstance, 176**

classes, *continued*
- **/understands?, 176**
- **unpromote, 164**

/classname, 175
/cleanoutclass, 165
cleanoutdict, 293
clearselection, 335
clearsendcontexts, 260
client debugging commands, 214
client-server interface, 2, 3, 179
- byte stream format, 323
- contacting the server, 211
- CPS facility, 3, 179
- downloading POSTSCRIPT language code, 179
- `psh`, 179
- supporting NeWS from other languages, 211
- system token list, 326
- token ordering, 323
- user token list, 326

ClientData, 247
clipcanvas, 260, 33, 265, 266
clipcanvaspath, 261, 33, 261
Color, 239
colorhsb, 304
Colormap, 240, 243
colormapentrytype dictionary keys, 242 *thru* 243
colormapentries
- **Colormap**, 243
- **Mask**, 243
- **Slot**, 243

colormapentrytype extension, 242
colormaps
- **Entries**, 242
- **Free**, 242
- **Installed**, 242
- **Visual**, 242

colormaptype dictionary keys, 242
colormaptype extension, 242
colorrgb, 304
colors, 5
- **contrastswithcurrent**, 261
- **currentcolor**, 264
- **hsbcolor**, 271
- **rgbcolor**, 282
- **RGBcolor**, 296
- **setcolor**, 283

colortype extension, 235
`compat.ps`, 292
conditional operators, 299
console, 308, 295
constants, 308
contacting the server
- for debugging, 213
- NEWSSERVER, 211

continueprocess, 261, 69, 287
contrastswithcurrent, 261
coordinate systems, 14
Coordinates, 249
copyarea, 261, 266
copying events, 84
counted objects, 222

counted objects, *continued*
counted references to, 222
reference tallies, 224
soft references, 223
uncounted references to, 222
counted references, 222
countfileinputtoken, 194
countfileinputtoken, 261
countinputqueue, 261
`cps` command, 180
CPS facility, 3, 179
argument types, 184
`cdef` command, 180, 183
`cdef` syntax, 183
`cdef`: asynchronous replies, 185, 188
`cdef`: requesting no reply, 185, 186
`cdef`: synchronous replies, 185, 186
closing the connection, 182
code example, 198
compiling the `.c` file, 180
connection example, 182
connection files, 181
connection utilities, 181
countfileinputtoken, 194
`cps` command, 180
the `.cps` file, 180, 183
creating equivalents for other languages, 211
debugging, 197
defining user tokens, 191
flushing the output buffer, 182
getfileinputtoken, 193
the `.h` file, 180
include files, 180
`ps_check_input()`, 190
`ps_peek_tag()`, 190
`ps_query_tag()`, 190
`ps_read_tag()`, 190
`ps_skip_input_value()`, 191
`ps_close_PostScript()`, 182
`ps_define_stack_token()`, 195
`ps_define_value_token()`, 195
`ps_define_word_token()`, 195
`ps_finddef()`, 195
`ps_flush_PostScript()`, 182
`ps_open_PostScript()`, 181
`ps_scaledef()`, 196
`ps_usetfont()`, 196
reading client's input connection, 190
setfileinputtoken, 193
tagprint, 187, 287
the `.cps` library, `libcps.a`, 180
typedprint, 187, 287
utilities for PostScript language operators, 191
`.cps` file, creating, 183
createcanvas, 293
createcolormap, 262
createcolorsegment, 262
createdevice, 262
createdevicecanvas, 55
createdevicecanvas, 293
createevent, 262, 83, 257, 267, 282
createmonitor, 74
createmonitor, 262, 274
createoverlay, 262, 41, 301
creating a new class, 139
creating a new instance, 156
creating canvases, 15
creating events, 83
cshow, 20
cshow, 304
current canvas
currentcanvas, 263
setting, 18
currentautobind, 263, 282
currentbackcolor, 263, 283
currentbackpixel, 263, 283
currentcanvas, 263
currentcolor, 264
currentcursorlocation, 264, 260
currentfontmem, 231
currentfontmem, 264, 284
currentinputfocus, 342
currentpacking, 299, 275
currentpath, 264, 236
currentpixel, 264, 284
currentplanemask, 264, 284
currentprintermatch, 264, 285
currentprocess!, 264, 60
currentrasteropcode, 265, 285
currentscreen, 345
currentshared, 294, 286
currentstate, 265, 235, 286
currenttime, 265
currenttimems, 265
Cursor, 51, 240
`cursor.ps`, 291, 302
CursorChar, 244
CursorColor, 244
CursorFont, 244
cursors, 51
building, 291
currentcursorlocation, 264
CursorChar, 244
CursorColor, 244
cursorfont, 291, 302
CursorFont, 244
fonts, 291
MaskChar, 244
MaskColor, 244
MaskFont, 244
setcursorlocation, 283
cursortype dictionary keys, 243 *thru* 244
cursortype extension, 243
cvad, 294
cvas, 294
cvis, 294

D

damage
clipcanvas, 260
damage events, 120

damage, *continued*
- **damagepath**, 265
- **eoextenddamage**, 266
- **extenddamage**, 267

damage, canvas, 13
damagepath, 265, 261
dbgbreak, 214, 216
dbgbreakenter, 216, 214
dbgbreakexit, 216, 214
dbgcall, 218
dbgcallbreak, 218
dbgcontinue, 217
dbgcontinuebreak, 217
dbgcopystack, 218
dbgenter, 218
dbgenterbreak, 217
dbgexit, 218
dbggetbreak, 218
dbgkill, 219
dbgkillbreak, 219
dbglistbreaks, 215
dbgmodifyproc, 219
dbgpatch, 219
dbgpatchbreak, 218
dbgprintf, 214, 216, 217, 296
dbgprintfenter, 216, 214
dbgprintfexit, 216, 214
dbgremove, 216
dbgremovebreak, 216
dbgstart, 215
dbgstop, 215
dbgwhere, 217
dbgwherebreak, 217
debugdict, 225
debugging, **213**, 6
- aliases, 219
- client commands, 214
- contacting the server, 213
- **dbgbreak**, 214
- **dbgbreakenter**, 216
- **dbgbreakexit**, 216
- **dbgcall**, 218
- **dgbcallbreak**, 218
- **dbgcontinue**, 217
- **dbgcontinuebreak**, 217
- **dbgcopystack**, 218
- **dbgenter**, 218
- **dbgenterbreak**, 214, 217
- **dbgexit**, 218
- **dbggetbreak**, 218
- **dbgkill**, 219
- **dbgkillbreak**, 219
- **dbglistbreaks**, 214, 215
- **dbgmodifyproc**, 219
- **dgbpatch**, 219
- **dbgpatchbreak**, 218
- **dbgprintf**, 214
- **dbgprintfenter**, 216
- **dbgprintfexit**, 216
- **dbgremove**, 216
- **dbgremovebreak**, 216

debugging, *continued*
- **dbgstart**, 215
- **dbgstop**, 215
- **dbgwhere**, 217
- **dbgwherebreak**, 217
- `debug.ps`, 213, 292
- errors, 215
- loading the debugger, 213
- miscellaneous hints, 219
- multiple connections, use of, 220
- multiple processes, 214
- starting the debugger, 213
- user commands, 215
- using the debugger, 213

default class, 161
/defaultclass, 162
defaulterroraction, 265, 76
DefineAutoLoads, 305
DefineAutoLoads, 305
deliverevent, 266
demos
- `rubber`, 301

/descendantof?, 176
/destroy, 165
destroying canvases, 22
destroying classes, 165
destroying instances, 165
device canvas, 15
dictbegin, 294
dictend, 294
dictionaries, 233
- **canvastype** keys, 7, 236 *thru* 241
- **nulloutdict**, 293
- **colormapentrytype** dictionary keys
- **colormaptype** keys, 242
- **cursortype** keys, 243 *thru* 244
- **dictbegin**, 294
- **dictend**, 294
- **environmenttype** keys, 244 *thru* 246
- **eventtype** keys, 79, 246, 249
- magic dictionaries, 233
- **nulloutdict**, 296
- **processtype** keys, 251 *thru* 255
- UnloggedEvents, 311
- utilities, 293, 296
- **visualtype** keys, 255 *thru* 256

dictionary
- size of **userdict**, 346

DictionaryStack, 251
dictkey, 294
distributed window system, 1
distribution of events, 85, 282
/doit, 153
drawing in canvases, 18

E

echo, 345
emptypath, 266
encodefont, 266
enter and exit events, 113
Entries, 242

environment objects
- **AutoRepeat**, 245
- **BellPercent**, 245
- **BellPitch**, 245
- **KeyClickPercent**, 245
- **Leds**, 245

environment variables
- **getenv**, 269
- HOME, 309
- OPENWINHOME, 309
- **putenv**, 279

environmenttype dictionary keys, 244 *thru* 246
environmenttype extension, 244
eoclipcanvas, 266, 34
eocopyarea, 266
eocurrentpath, 266
eoextenddamage, 266
eoextenddamageall, 266, 267
eoreshapecanvas, 267
eowritecanvas, 267, 288, 289
eowritescreen, 267, 267, 288, 289
$error, 252
ErrorCode, 252
ErrorDetailLevel, 253
errordict, 252
errors
- accept, 252
- dictfull, 252
- dictstackoverflow, 252
- dictstackunderflow, 252
- execstackoverflow, 252
- interrupt, 252
- invalidaccess, 252
- invalidexit, 252
- invalidfileaccess, 252
- invalidfont, 252
- invalidrestore, 252
- ioerr, 252
- killprocess, 252
- limitcheck, 252
- nocurrentpoint, 252
- none, 252
- rangecheck, 252
- stackoverflow, 252
- stackunderflow, 252
- syntaxerror, 252
- typecheck, 252
- undefined, 252
- undefinedfilename, 252
- undefinedresult, 252
- unimplemented, 252
- unmatchedmarck, 252
- unregistered, 252
- VMerror, 252

event dictionary keys, 79
event distribution, 80, 96
event operators, 80
eventlog, 311
eventmgrinterest, 300
events, 79, 91
- **Action**, 91, 92, 247

events, *continued*
- as event dictionaries, 246
- **awaitevent**, 85, 257
- awaiting events, 85
- basic event operations, 83
- blocking the event queue, 122, 124
- **blockinputqueue**, 122, 258
- **Canvas**, 97
- canvas crossing events, 113
- canvas event consumption, 99
- canvas interest lists, 96
- **Canvas**, 92, 96, 247
- changing and reusing, 84
- **ClientData**, 91
- **ClientData**, 247
- **Coordinates**, 87
- **Coordinates**, 249
- copying, 84
- **createevent**, 83, 262
- creating events, 83
- damage events, 120
- dictionaries for **Name**, **Action**, and **Canvas**, 92
- distribution of, 80, 85, 96, 282
- enter and exit events, 113
- event dictionary keys, 79
- event operators, 80
- **eventlog**, 292
- **eventmgrinterest**, 300
- **EventsConsumed** key (in **canvastype** dictionary), 99
- exclusive interests, 100
- **Exclusivity**, 97
- **Exclusivity**, 100, 247
- executable matches, 92
- **Exclusivity**, 93
- expressing interests, 84
- **expressinterest**, 84, 267
- focus events, 118
- **forkeventmgr**, 300
- **getcompatinputdist**, 126
- **geteventlogger**, 128, 269
- global event queue, 81
- input events, 79
- interest list order, 97
- interest matching order, 97
- **Interest**, 247
- interests, 79, 92, 96
- **Interests** key (in **processtype** dictionary), 253
- **IsInterest**, 247
- **IsPreChild**, 247
- **IsQueued**, 248
- keyboard events, 109
- **KeyState**, 248
- **lasteventtime**, 273
- **lasteventtime**, 273
- local event queue, 82
- location of, 87
- logging, 310
- logging events, 128
- matching multiple interests, 96
- matching multiple interests, example, 102
- mouse events, 110
- **Name**, 91, 92, 248
- obsolescence, 223
- obsolescence events, 109

events, *continued*
- post-match processing, 92
- pre-child and post-child, 96, 102
- **PreChild**, 97
- **Priority**, 97, 248
- process died events, 109
- process interests list, 96
- **Process**, 92, 248
- process-generated, 79
- **recallevent**, 91
- **recallevent**, 279
- recalling events, 91
- **redistributeevent**, 100, 280
- redistribution, 100
- **revokeinterest**, 91, 281
- rules for matching events to interests, 91
- **sendevent**, 85, 282
- sending events into distribution, 85
- **Serial**, 92, 248
- **setcompatinputdist**, 126
- **seteventlogger**, 128, 283
- specifying additional information, 91
- stopping distribution of an event, 99
- synchronization, 82, 122
- **Synchronous**, 124
- **Synchronous**, 249
- system-generated, 79, 109
- time of distribution, 89
- **TimeStamp**, 81, 85, 89, 249, 273
- **TimeStampMS**, 249
- **unblockinputqueue**, 123
- unlogging, 311
- **XLocation**, 87
- **XLocation**, 249
- **YLocation**, 87
- **YLocation**, 249

EventsConsumed, 239
eventtype dictionary keys, 246 *thru* 249
environmenttype extension, 246
Exclusivity, 247
Execee, 253
executable matches, 92
executeonly, 345
execution stack
- maximum nesting depth, 346

ExecutionStack, 253
expressing interests, 84
expressinterest, 267, 84, 257, 262, 280, 281, 282
extenddamage, 267
extenddamageall, 267
extended input, **329**
- **addeditkeysinterest**, 332
- **addfunctionnamesinterest**, 332
- **addfunctionstringsinterest**, 332
- **addkbdinterests**, 331
- **addselectioninterests**, 333, 335
- **ascii_keymap**, 331
- **awaitevent**, 331
- buffer model for selection, 333
- **clearselection**, 335
- **/ClickFocus**, 342
- communication model for selection, 333

extended input, *continued*
- **ContentsAscii**, 334
- **ContentsPostScript**, 334
- **currentinputfocus**, 342
- cursor control, 332
- **/CursorFocus**, 342
- **/DefaultFocus**, 343
- **/DeSelect**, 339
- **/ExtendSelectionTo**, 338
- focus events, 331
- function key assignment, 332
- function keys, 331
- **/FunctionL7**, 332
- **getselection**, 336
- **hasfocus**, 342
- input focus, 340
- **/InsertValue**, 331, 332, 336
- keyboard editing, 332
- keyboard input, 331
- **Rank**, 335
- **revokekdbinterests**, 331
- selection events, 336
- selection procedures, 335
- **SelectionHolder**, 334
- **SelectionLastIndex**, 334
- **SelectionObjsize**, 334
- **selectionrequest**, 336
- **/SelectionRequest**, 340
- **SelectionRequester**, 335
- **SelectionResponder**, 334
- **selectionresponse**, 336, 340
- **/SelectionResponse**, 336
- selections, 333
- **SelectionStartIndex**, 334
- **setfocusmode**, 342
- **setinputfocus**, 342
- **setselection**, 336
- **/SetSelectionAt**, 337
- **/ShelveSelection**, 339
- **/UnknownRequest**, 336
- **ViewPoint**, 330

extensibility, 6
extensions in NeWS
- operators, 257
- types, 233

F

file
- access utilities, 305
- raster, 279

file, 268
filepathopen, 306
filepathparse, 306
filepathrun, 306
files
- number allowed open, 346
- POSTSCRIPT language extensibility, 6, 213
- reading and writing canvases to, 44

fillcanvas, 304, 18
findfilefont, 268
focus events, 118
font object, 268
font support, 5

font utilities
- **cvas**, 294
- **cvis**, 294
- **findfilefont**, 268
- **stringbbox**, 297

fontascent, 295, 297
fontdescent, 295, 297
fontheight, 295, 297
fontpath, 308
fonts
- cache, 231
- **cursorfont**, 291
- **PrinterMatched**, 250
- **WidthArray**, 250

fonttype extension, 249
fork!, 268, 62, 250, 288
forkeventmgr, 300
fprintf, 295, 308
framebuffer canvas, 10, 15
framebuffer, 308
framedevice, 345
Free, 242
function keys
- assigning, 309
- repeating, 310

G

getanimated, 301
getcanvaslocation, 268, 22, 274
getcanvasshape, 268
getcard32, 269, 278
getclick, 301
getcolor, 269, 279
getcompateventdist, 269, 283
getcompatinputdist, 126
getenv, 269, 279
geteventlogger, 128
geteventlogger, 269, 284
getfileinputtoken, 193
getfileinputtoken, 269
getkeyboardtranslation, 270, 272, 284
getprocesses, 72
getprocesses, 270
getprocessgroup, 270, 71
getrect, 301
getselection, 336
getsocketlocaladdress, 270
getsocketpeername, 270, 257
getwholerect, 302
global event queue, 81
globalroot, 270
Grabbed, 241
GrabToken, 241
graphics states
- **currentstate**, 235, 265
- **setstate**, 235, 286

graphics utilities, 303
graphicsstatetype extension, 235
growabledict, 295

H

harden, 224
harden, 271, 287
hasfocus, 342
HOME, 309
HOME, 309
hsbcolor, 271, 235

I

imagecanvas, 271, 259, 272, 279
imagemaskcanvas, 272, 49, 259, 271
imagepath, 272
imaging, 4, 50
- **buildimage**, 258
- **imagecanvas**, 47, 271
- **imagemaskcanvas**, 272

implementation limits, 346
`init.ps`, 291
- size of **userdict**, 346

initclip, 280
input, *see* events
input events, 79
insertcanvasabove, 272, 37, 260
insertcanvasbelow, 272, 37, 260
insetrect, 303
insetrrect, 304
Installed, 242
/installmethod, 153
/instanceof?, 176
Interest, 247
interests, 79, 253
- **Canvas**, 96
- canvas interest lists, 96
- exclusive interests, 100
- executable matches, 92
- expressing, 84
- interest list order, 97
- matching multiple, 96
- matching multiple, example, 102
- **Name**, **Action**, and **Canvas** keys, 92
- pre-child and post-child, 96, 102

Interests, 239, 253
interprocess communication, 2
intrinsic class, 161
isarray?, 299, 272
isclass?, 175
isinstance?, 175
IsInterest, 247
isobject?, 175
IsPreChild, 247
IsQueued, 248

J

journalend, 307
journalling, 307
- controls, 307
- internal variables, 307
- **journal**, 308
- `journal.ps`, 292
- **journalend**, 307

journalling, *continued*
- **journalplay**, 307
- **journalrecord**, 307
- **PlayBackFile**, 307
- **PlayForever**, 308
- **RecordFile**, 307
- **State**, 308

journalplay, 307
journalrecord, 307

K

key mapping, 309
keyboard
- repeating keys, 287, 310
- **stoprepeating**, 287

keyboard events, 109
keyboardtype, 272, 270, 284
KeyClickPercent, 245
keys
- **bindkey**, 309
- mapping, 309
- `repeat.ps`, 292
- repeating, 310
- **unbindkey**, 310

KeyState, 248
killprocess, 272, 68, 268
killprocessgroup, 71
killprocessgroup, 273, 268

L

lasteventkeystate, 273
lasteventtime, 273
lasteventtimems, 273
lasteventx, 273
lasteventy, 273
Leds, 245
lightweight processes, 2
list of class methods, 177
list of class operators, 177
liteUI user interface package, 330
`liteUI.ps`, 329
litstring, 295
LoadFile, 306
local event queue, 82
localhostname, 308
localhostnamearray, 273
location of events, 87
logging events, 128, 310
- `eventlog.ps`, 292

M

magic dictionaries, 233
Mapped, 17, 238
mapping canvases, 17
Mask, 243
MaskChar, 244
MaskColor, 244
MaskFont, 244
/MatchedEvents, 239
matching multiple interests, 96
max, 273
memory management, **221**
- counted objects, 221, 222
- **currentfontmem**, 231
- **debugdict**, 225
- **harden**, 224
- object types, 222
- **objectdump**, 225
- obsolescence events, 223
- operators, 224
- operators for debugging, 225
- **refcnt**, 226
- reference counting, 221
- reference tallies, 224
- **reffinder**, 227
- **setfontmem**, 231
- soft references, 223
- **soft**, 225
- **soften**, 224
- uncounted objects, 221, 222
- **vmstatus**, 228

method compilation, **150**, 147, 149
/methodcompile, 152
min, 274
minim, 308
missing POSTSCRIPT language operators, 345
modifying the server
- saving keystrokes, 219
- `.startup.ps`, 292
- `.user.ps`, 292

modifyproc, 296
monitor, 74
- **createmonitor**, 262

monitor, 274, 262
monitorlocked, 74
monitorlocked, 274, 262, 274
monitortype extension, 235
mouse, see *cursor*
mouse events, 110
movecanvas, 274, 21, 268, 272
moving canvases, 20
multiple inheritance of classes, 166 *thru* 174

N

/name, 174
Name, 248
/new, 157
newcanvas, 274, 15, 236
newcursor, 275, 53, 243
/newdefault, 161
/newinit, 158, 157
/newmagic, 159
/newobject, 157, 157
newprocessgroup, 275, 70, 273
NeWS
- classes, 6
- client-server interface, 3
- color support, 5
- debugging, 6, 213
- distributed window system, 1
- font support, 5

NeWS, *continued*
memory management, 221
operators, 1, 2
POSTSCRIPT language extension files, 6
types, 1
NeWS procedure files, *see* *.ps files
NEWSSERVER, 211
newsserverstr, 211
noaccess, 345
/NoEvents, 239
non-retained canvases, 13
nulldict, 308
nulloutdict, 296
nullproc, 308
nullstring, 309
number
limits, 346

O

objectdump, 225
objectdump, 275
objects
types in memory management, 222
obsolescence events, 109, 223
/obsolete, 166
obsolete classes, 165
obsolete instances, 165
omitted POSTSCRIPT language operators, 345
opaque canvases, 22
OPENWINHOME, 309
OPENWINHOME, 309
openwinversion, 309
operand stack
size limit, 346
OperandStack, 254
operators, 313
arrays, 297
conditional, 299
extensions in NeWS, 1
memory management, 224
memory management debugging, 225
NeWS extensions, 2
ovalframe, 304
ovalpath, 304
overlay canvases, 40, 262
createoverlay, 41, 262
getanimated, 301
OverrideRedirect, 240
overriding class variables, 162

P

packedarray, 275, 299
PackedArrays, 254
packedarraytype extension, 236
Parent, 39, 238
pathforallvec, 275
paths, 236
currentpath, 236
setpath, 236
pathtype extension, 236
pause!, 276, 64
pipe, 276
PlayBackFile, 307
PlayForever, 308
pointinpath, 276
points2rect, 303
polyline, 304
polypath, 304
polyrectline, 304
polyrectpath, 305
portability
retained, 274
post-match processing of events, 92
postcrossings, 276
POSTSCRIPT language, 1
extensibility, 1
extension files, 6
POSTSCRIPT language extensions
in *.ps files, 292
printer compatability
NeWS?, 291
statusdict, 291
test to see if running under NeWS, 291
PrinterMatched, 250
printermatchfont, 278
printername, 291
printf, 296, 214
priority of events, 97
Priority, 248, 254
process died events, 109
process interests list, 96
Process, 248
process-generated events, 79
processes, 57
BindOverride, 251
breakpoint, 69, 258
continueprocess, 69, 261
createmonitor, 74
current execution state names, 254
currentprocess!, 60, 264
DictionaryStack, 251
$error, 252
ErrorCode, 252
ErrorDetailLevel, 253
errordict, 252
Execee, 253
ExecutionStack, 253
fork!, 62, 268
getprocesses, 72
getprocessgroup, 71
Interests, 253
killprocess, 68, 272
killprocessgroup, 71, 273
lightweight, 2
monitor, 74
monitorlocked, 74
newprocessgroup, 70, 275
OperandStack, 254
PackedArrays, 254
pause!, 64, 276
Priority, 254

processes, *continued*
- **ProcessName**, 254
- **runprogram**, 282
- scheduling policy of, 254
- **SendContexts**, 255
- **SendStack**, 255
- **sleep**, 66
- **State**, 254
- **Stderr**, 255
- **Stdout**, 255
- **suspendprocess**, 69, 287
- **waitprocess!**, 65, 288

ProcessName, 254
processtype dictionary keys, 251 *thru* 255
processtype extension, 250
promote, 163
promoted?, 164
promoting class variables, 163
`*.ps` files, **291**
- `basics.ps`, 291
- `class.ps`, 292
- `compat.ps`, 292
- `cursor.ps`, 291, 302
- `debug.ps`, 213, 292
- `eventlog.ps`, 292
- `init.ps`, 291, 346
- `journal.ps`, 292
- `liteUI.ps`, 329
- POSTSCRIPT procedures they define, 292
- `redbook.ps`, 291
- `repeat.ps`, 292
- `.startup.ps`, 292, 345
- `statdict.ps`, 291
- `.user.ps`, 219, 292, 345
- `util.ps`, 292, 346

`ps_check_input()`, 190
`ps_peek_tag()`, 190
`ps_query_tag()`, 190
`ps_read_tag()`, 190
`ps_skip_input_value()`, 191
`ps_arc(x,y,r,a0,a1)`, 191
`ps_closepath()`, 191
`ps_close_PostScript()`, 182
`ps_cshow(cstring s)`, 191
`ps_define_stack_token()`, 195
`ps_define_value_token()`, 195
`ps_define_word_token()`, 195
`ps_fill()`, 191
`ps_finddef()`, 195
`ps_findfont(string font)`, 191
`ps_flush_PostScript()`, 182
`ps_grestore()`, 191
`ps_gsave()`, 191
`psh`, 179
`ps_lineto(x,y)`, 191
`ps_moveto(x,y)`, 191
`ps_open_PostScript()`, 181
`ps_rlineto(x,y)`, 191
`ps_rmoveto(x,y)`, 191
`ps_scaledef()`, 196
`ps_scalefont(n)`, 191
`ps_setfont()`, 191
`ps_show(string s)`, 191
`ps_stroke()`, 191
`ps_usetfont()`, 196
putcard32, 278, 269
putcolor, 279, 269
putenv, 279, 269

Q

quicksort, 299

R

random, 296
raster files, 279
rasteropcode
- **currentrasteropcode**, 265
- **setrasteropcode**, 285

readcanvas, 279, 47, 271, 272
recallevent, 91
recallevent, 279, 282
recalling events, 91
RecordFile, 307
rect, 303
rect2points, 303
rectangle utilities, 303
rectframe, 305
rectpath, 303, 16
rectsoverlap, 303
`redbook.ps`, 291
redef, 140
redistributeevent, 280, 100, 257, 262, 267, 282
refcnt, 226
refcnt, 280
reference counting, 221
- counted objects, 221, 222
- uncounted objects, 221, 222

reference tallies for counted objects, 224
reffinder, 227
reffinder, 280
refork, 296
resetfile, 345
reshapecanvas, 280, 16, 267, 274
reshaping canvases, 16
retain threshold, 29
retained
- portability, 274

retained canvases, 13, 29
Retained, 238
reversepath, 345
revokeinterest, 91
revokeinterest, 281, 267, 302
revokekbdinterests, 331
revoking interests, 91
rgbcolor, 282, 235, 271, 296
RGBcolor, 296
RowBytes, 241
rrectframe, 305
rrectpath
- **rrectpath**, 29

rrectpath, 305, 304
rshow, 305
`rubber`, 301
rules for matching events to interests, 91
runprogram, 282

S

SaveBehind, 32, 239
saving behind canvases, 32
selectionrequest, 336
selectionresponse, 336
self, 145 *thru* 150
send, 140 *thru* 145
send context, 140
send, 282, 260
sendcidevent, 307
SendContexts, 255
sendevent, 282, 85, 257, 258, 262, 267, 279
sending events into distribution, 85
sendstack, 296
SendStack, 255
/sendtopmost, 177
Serial, 248
setautobind, 282, 263
setbackcolor, 283, 263
setbackpixel, 283, 263
setcanvas, 283, 18
setcolor, 283, 296
setcompateventdist, 283, 269
setcompatinputdist, 126
setcursorlocation, 283
seteventlogger, 128
seteventlogger, 283, 269
setfileinputtoken, 193
setfileinputtoken, 284
setfocusmode, 342
setfontmem, 231
setfontmem, 284, 264
sethsbcolor, 283
setinputfocus, 342
setkeyboardtranslation, 284, 270, 272
SetLocalDicts, 154
/setname, 174
setpacking, 299, 275, 299
setpath, 284, 236
setpixel, 284
setplanemask, 284, 264
setprintermatch, 285, 264
setrasteropcode, 285, 265
setretainthreshold, 286
setrgbcolor, 283
setsccsbatch, 291
setscreen, 345
setselection, 336
setshade, 305
setshared, 286, 294
setstandardcursor, 302
setstate, 286, 235, 265
setsysinputtoken, 286
SharedFile, 240
showpage, 345
shutdownserver, 286
sibling canvases, 12, 36
Size, 255
sleep, 297, 66
Slot, 243
sockets
    getsocketlocaladdress, 270
    getsocketpeername, 270
soft, 225
soft references, 223
soft, 286
soften, 224
soften, 287, 271
sprintf, 297
start, 345
starting the debugger, 213
startkeyboardandmouse, 287
`statdict.ps`, 291
State, 254, 308
statusdict, 291
Stderr, 255
Stdout, 255
stencil/paint model, 4
stoprepeating, 287
stringbbox, 297
strings
    length limit, 346
strokecanvas, 305
/subclasses, 176
/SubClassResponsibility, 162
SubstructureRedirect, 240
super, 145 *thru* 150
/superclasses, 175
suspendprocess, 287, 69, 261
synchronization of events, 82, 122
synchronous replies from server, 185, 186
Synchronous, 249
system token list, 326
system-generated events, 79, 109
SystemPriority, 254

T

tagprint, 187, 327
tagprint, 287
time of event distribution, 89
time values
    resolution, 249
TimeStamp, 81, 85, 89, 249, 273
TimeStampMS, 249
token encoding, 323
token lists, 325
tokenization, 191
tokens, 191
TopCanvas, 36, 237
TopChild, 36, 238

/topmostdescendant, 176
/topmostinstance, 176
transparent canvases, 22
Transparent, 22, 238
truetype, 287
typedprint, 187, 327
typedprint, 287
types, **233**
 as magic dictionaries, 233
 canvastype, 7, 236
 colormapentrytype, 242
 colormaptype, 242
 colortype, 235
 cursortype, 243
 environmenttype, 244
 eventtype, 79, 246
 extensions in NeWS, 1, 235
 fonttype, 249
 graphicsstate, 235
 graphicsstatetype, 235
 memory management objects, 222
 monitortype, 235
 packedarraytype, 236
 pathtype, 236
 processtype, 250
 standard types, 234
 visualtype, 255

U

unbindkey, 310
unblockinputqueue, 288, 123, 258
uncounted objects, 221, 222
uncounted references, 222
undef, 288
/understands?, 176
uniquecid, 307
unlogging, 311
unpromote, 164
unrooted canvases, 14
user debugging commands, 215
user token list, 326
user tokens, 191
 declaring, 195
 defining, 195
`.user.ps`, 219, 292
UserPriority, 254
UserProfile, 330
using the debugger, 213
`util.ps`
 size of **userdict**, 346
utilities
 miscellaneous, 293
 `util.ps`, 292

V

version, 309
VisibilityInterest, 240
Visual, 240, 242
VisualList, 240
visuals
 BitsPerPixel, 256

visuals, *continued*
 Class, 255
 Size, 255
visualtype dictionary keys, 255 *thru* 256
visualtype extension, 255
vmstatus, 228
vmstatus, 288

W

waitprocess!, 288, 65, 268, 301, 302
WidthArray, 250
writecanvas, 288, 45, 259, 267, 279, 289
writeobject, 289
writescreen, 289, 45, 267, 288

X

X11/NeWS server, 2
 lightweight processes, 2
XID, 240
XLocation, 249

Y

YLocation, 249

Notes

Notes

Notes

Notes

Notes

Notes

Notes