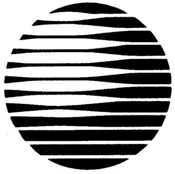
UNIX® SYSTEM V
RELEASE 4

Programmer's Guide: X11/NeWS®
Graphical Windowing System
tNt Technical Reference Manual

UNIX Software Operation



UNIX® SYSTEM V RELEASE 4

Programmer's Guide: X11/NeWS® Graphical Windowing System tNt Technical Reference Manual

UNIX Software Operation

Copyright 1990, 1989, 1988, 1987, 1986, 1985, 1984, 1983 AT&T
All Rights Reserved
Printed in USA

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

No part of this publication may be reproduced or transmitted in any form or by any means—graphic, electronic, electrical, mechanical, or chemical, including photocopying, recording in any medium, taping, by any computer or information storage and retrieval systems, etc., without prior permissions in writing from AT&T.

ACKNOWLEDGEMENT

Parts of this book are reproduced with the permission of the following organizations: Sun Microsystems, Inc., Digital Equipment Corporation (DEC), X Window System is a trademark of Massachusetts Institute of Technology, X11 is a trademark of Massachusetts Institute of Technology, X11/NeWS is a registered trademark of Sun Microsystems, Inc.

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, AT&T assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. AT&T further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. AT&T disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, *including implied warranties of merchantability or fitness for a particular purpose*. AT&T makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

AT&T reserves the right to make changes without further notice to any products herein to improve reliability, function, or design.

TRADEMARKS

The Sun logo, Sun Microsystems, and Sun Workstations are registered trademarks of Sun Microsystems, Inc.
Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunInstall, SunOS, Sun View, NFS, NeWS and SPARC are trademarks of Sun Microsystems, Inc.
PostScript is a registered trademark of Adobe Systems
UNIX is a registered trademark of AT&T

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-931858-5

UNIX
PRESS
A Prentice Hall Title

P R E N T I C E   H A L L

ORDERING INFORMATION

UNIX® SYSTEM V, RELEASE 4 DOCUMENTATION

To order single copies of UNIX® SYSTEM V, Release 4 documentation, please call (201) 767-5937.

ATTENTION DOCUMENTATION MANAGERS AND TRAINING DIRECTORS:
For bulk purchases in excess of 30 copies please write to:
Corporate Sales
Prentice Hall
Englewood Cliffs, N.J. 07632
Or call: (201) 592-2498

ATTENTION GOVERNMENT CUSTOMERS: For GSA and other pricing information please call (201) 767-5994.

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

AT&T UNIX® System V Release 4

General Use and System Administration

UNIX® System V Release 4 Network User's and Administrator's Guide
UNIX® System V Release 4 Product Overview and Master Index
UNIX® System V Release 4 System Administrator's Guide
UNIX® System V Release 4 System Administrator's Reference Manual
UNIX® System V Release 4 User's Guide
UNIX® System V Release 4 User's Reference Manual

General Programmer's Series

UNIX® System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools
UNIX® System V Release 4 Programmer's Guide: Character User Interface (FMLI and ETI)
UNIX® System V Release 4 Programmer's Guide: Networking Interfaces
UNIX® System V Release 4 Programmer's Guide: POSIX Conformance
UNIX® System V Release 4 Programmer's Guide: System Services and Application Packaging Tools
UNIX® System V Release 4 Programmer's Reference Manual

System Programmer's Series

UNIX® System V Release 4 Device Driver Interface / Driver–Kernel Interface (DDI / DKI) Reference Manual
UNIX® System V Release 4 Programmer's Guide: STREAMS

Migration Series

UNIX® System V Release 4 ANSI C Transition Guide
UNIX® System V Release 4 BSD / XENIX® Compatibility Guide
UNIX® System V Release 4 Migration Guide

Graphics Series

UNIX® System V Release 4 OPEN LOOK™ Graphical User Interface Programmer's Reference Manual
UNIX® System V Release 4 OPEN LOOK™ Graphical User Interface User's Guide
UNIX® System V Release 4 Programmer's Guide: OPEN LOOK™ Graphical User Interface
UNIX® System V Release 4 Programmer's Guide: X11/NeWS® Graphical Windowing System NeWS
UNIX® System V Release 4 Programmer's Guide: X11/NeWS® Graphical Windowing System Server Guide
UNIX® System V Release 4 Programmer's Guide: X11/NeWS® Graphical Windowing System tNt Technical Reference Manual
UNIX® System V Release 4 Programmer's Guide: X11/NeWS® Graphical Windowing System XVIEW™
UNIX® System V Release 4 Programmer's Guide: XWIN™ Graphical Windowing System Addenda: Technical Papers
UNIX® System V Release 4 Programmer's Guide: XWIN™ Graphical Windowing System The X Toolkit
UNIX® System V Release 4 Programmer's Guide: XWIN™ Graphical Windowing System Xlib – C Language Interface

Available from Prentice Hall

Contents

1 Introduction 1-1

2 The Wire Service 2-1

3 Canvases 3-1

4 Managing Groups of Canvases 4-1

5 Menus and Other Selection Lists 5-1

6 Controls 6-1

7 Graphics 7-1

8 The NeWS Development Environment Input Model 8-1

9 Selections 9-1

10 Miscellaneous Topics 10-1

11 Interface Reference 11-1

Figures and Tables

Figure 4-1: Bags 4-1
Figure 4-2: Frame Hierarchy 4-14
Figure 4-3: Top Down Coordinates 4-30
Figure 4-4: Bottom Up Coordinates 4-30
Figure 4-5: Compass Point Notation 4-32

PREFACE

PREFACE

Preface

Preface 1
What's in the Chapters 3

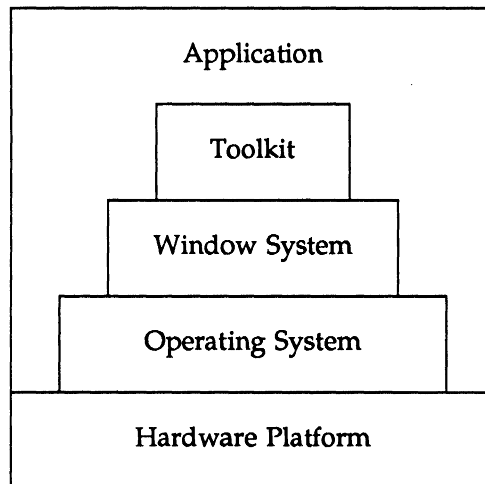
Preface

tNt extends X11/NeWS® in two areas, offering:

- classes of objects that implement the OPEN LOOK™ Graphical User Interface, and
- the Wire Service, an enhanced means of communication between the server and a client program.

To see what this means to you, let's put it in perspective, beginning with the generic model of a window system shown below.

Figure 1: Generic Window Architecture



Starting from the hardware, each layer of software manages resources and extends the capabilities of the system.

The operating system manages hardware resources, and provides services to access them. The type of management and service can vary widely. Simple operating systems manage the hardware resources, sophisticated operating systems extend those resources. For example, even primitive operating systems include a file system of some sort. The file system takes raw mass storage and turns it into files and directories of files. More sophisticated operating systems (certainly anything that supports X11/NeWS, such as UNIX® offer multitasking,

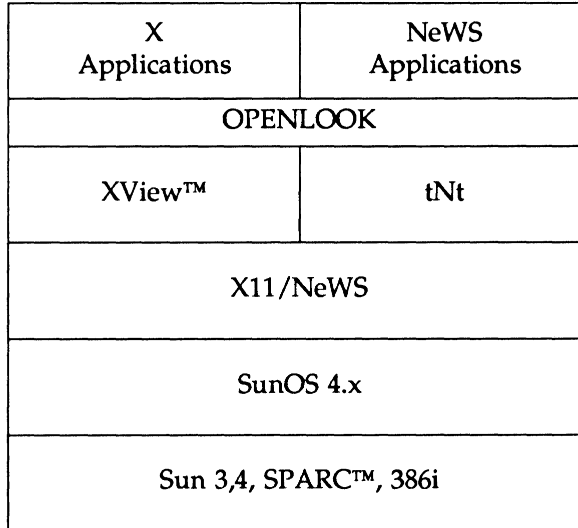
which does the same thing for the raw processor/memory resource, turning it into multiple jobs apparently executing at the same time.

The next layer up in the generic model, the window system, applies this same philosophy to the display resource. The window system takes a raw high resolution display and turns it into multiple virtual displays called windows.

The window system toolkit provides another level of service in allowing you to manupulate the appearance and function ("look and feel") of the windows. The toolkit provides varying degrees of assistance to the application in defining the interface the application's user will see. In some cases the toolkit offers assorted user interface components for the application builders to use as they see fit. In other cases, the toolkit implements (and enforces) a standard user interface to be used by all applications.

The situation with X11/NeWS is shown in Figure *xnewsarch* below. What is unique to X11/NeWS is that there are two toolkits, one for X11™ applications and one for NeWS applications. Both toolkits implement the emerging OPEN LOOK graphical user interface standard.

Figure 2: X11/NeWS Architecture



This manual does not address the X Window™ side of X11/NeWS, but it is important to remember that it is there because you will occasionally run across things that owe their existence to the X11 side.

What's in the Chapters

The chapters are:

Chapter 1, "Introduction" is a brief introduction and includes information on running tNt demos.

Chapter 2, "The Wire Service" explains the tNt's enhancement to NeWS' support for client-server communications.

Chapter3, "Canvases" provides details on ClassCanvas which provides the basic underpinning for windows, menus, and controls.

Chapter 4, "Managing Groups of Canvases" explains about bags. When you need to bring a group of canvases together to accomplish some task you'll use a bag to manage them. This chapter also contains information about managing the input focus.

Chapter 5, "Menus and Other Selection Lists" explains how tNt enables you to build menus quickly and associate them with canvases.

Chapter 6, "Controls" explains about the different kinds of controls tNt provides. Some kinds of controls are scroll bars, sliders, dials, and buttons.

Chapter 7, "Graphics" contains information on the tNt facility for providing images for buttons, menus, controls and labels without the overhead of using canvases.

Chapter 8, "The NeWS Development Environment Input Model" is a comprehensive, in-depth look at the facilities tNt has added to the NeWS system for handling user input.

Chapter 9, "Selections" explains how tNt handles user selection of objects like text and windows.

Chapter 10, "Miscellaneous Topics" has information on ClassTarget, tNt's class that helps manages references to other objects and helps prevent dangling references. This chapter also contains information that helps to define what an tNt application is.

Chapter 11, "Interface Reference" contains the details of the function calls of the Wire Service and the method interface for selected tNt classes.

For further information on the OPEN LOOK Graphical User Interface, please consult:

- *OPENLOOK Functional Specification 1.0* Sun Microsystems Inc., part number 800-3355-05.
- *OPEN LOOK UI Style Guide,* Sun Microsystems Inc., part number 800-3356-06.

For information on the X11/NeWS server, see:

- *NeWS Programmer's Guide,* Sun Microsystems Inc., part number 800-2379-04.

For information on the PostScript® Language, see:

- *POSTSCRIPT Language Reference Manual,* Adobe Systems Inc., Addison Wesley, 1985, ISBN 0-201-10174-2.

1. INTRODUCTION

1. INTRODUCTION

1 Introduction

Introduction 1-1
Philosophy 1-1
Learning the System 1-1
Conventions 1-2

Introduction

The NeWS Development Environment provides a powerful and flexible extension to the NeWS window system. It is object-oriented in design and implements much of the look and feel described in the OPEN LOOK UI specification.

The toolkit is composed of two pieces: a set of PostScript classes which implement parts of the OpenLook specification, and completely replaces the Lite toolkit; and a C library, called the Wire Service, which significantly enhances CPS as the client/server communications package.

The two parts of the toolkit are quite independent. You can use only the PostScript components in a server-resident program, or even communicate with a client process via unadorned CPS. Similarly, the Wire Service does not rely on the PostScript classes at all, and could theoretically be used with any other server-side toolkit.

Philosophy

The philosophy behind the PostScript-based portion of the NeWS Development Environment is to provide a useful set of OPENLOOK components, underpinned by a powerful and flexible set of intrinsic classes that are not specific to any particular user interface.

The Wire Service is intended to provide the lowest common denominator in client-server communications needs. It handles registration of client-side callbacks, a notification mechanism by which to call them, and a synchronization system for server/client communication. It is almost transparent in its simplicity.

Learning the System

As with any powerful system, the NeWS Development Environment takes some effort to learn. To help you learn to get all the power out of the toolkit, demonstration code has been built directly into the the class hierarchy. This comes in the form of **/demo** methods in many of the classes. However, in order to decrease the size of the code that the toolkit loads into the server, the demo methods are not available by default. To enable them you must put the following line in the .user.ps file in your home directory.

```
systemdict /IncludeDemos? true put
```

Two files, bag-example.ps and selections-example.ps have been included on the tape that includes this documentation. Once you've started the server you can execute these examples via psh(1). Complete listing of the code for these example can be found in the following chapters:

bag-example.ps	Chapter 4. Managing Groups of Canvases
selections-example.ps	Chapter 8. Selections

Conventions

Several conventions are used in this manual:

- Toolkit methods are in **bold**.
- PostScript operators are in **helvetica-bold**.
- Code examples are in 10-point helvetica.
- At the beginning of major sections the relevant portion of the toolkit's class tree is reproduced to provide a connection between the subject being discussed and its location in the class hierarchy.

2. WIRE SERVICE

2. WIRE SERVICE

2 The Wire Service

The Wire Service 2-1
Purpose of the Wire Service 2-1
An overview of the components 2-1
- The Notifier 2-1
- Connection Management 2-1
- Resource Allocation 2-2
- Server-client synchronization 2-2

Notification 2-2
Connection Management 2-5
Resource Allocation 2-8
- Tag Allocation 2-8
- Token Allocation 2-9

Server-Client synchronization 2-11
Building a typical application 2-12

The Wire Service

Purpose of the Wire Service

The NeWS Wire Service is a server-client communications package that provides support and management of server connections, a client notifier, and shared resources. The Wire Service has four primary components: a notifier, a connection manager, a tag/token allocator, and a server-client synchronization mechanism.

Connection management enables a client to initiate multiple connections to one or more servers. Allocation of handles allows the C language to reference PostScript objects in the server (Tokens) and the PostScript language to reference C objects in the client (Tags). Event notification associates client procedures with specific events. When the event is received, the notifier will execute the appropriate procedure. Synchronization permits a server process to pause and wait for notification from a client process.

An overview of the components

The Notifier

The Notifier provides a lightweight mechanism for allowing clients to register callback functions associated with asynchronous messages from the server(s). Using the tag allocator, a client reserves a range of tags and associates a C procedure with each tag. When a tag is read from one of the server connections, the Notifier invokes the appropriate procedure. The notifier supports both applications that wish to "own" control of the main loop construct (e.g., MacIntosh style applications) and those that wish to be solely callback driven (e.g., Sunview style).

Connection Management

Connection management is the second major component of the Wire Service. An application can open a single connection to a NeWS server or multiple connections to a single server or multiple servers. A connection is abstracted to the notion of a Wire and facilities are provided for various forms of manipulation (opening, closing, temporarily disabling, etc.). Events from the server come back on the wire and are read using normal CPS operations (see the *NeWS Programmer's Guide*).

Resource Allocation

There are two resource allocators in the Wire Service: a tag allocator and a token allocator. While they have parallel architectures, they have complementary purposes. Tags are used to keep a handles toC procedures that can be used in the server, and tokens are handles to server objects for client manipulation.

Tags are allocated individually or in ranges on a per-client basis. Thus a single tag/callback pair that is registered with the Notifier can be used from multiple wires (and hence multiple servers). To provide compatibility with tag usage in applications written with statically defined tags, a range of tags can be reserved and removed from the allocator.

Tokens are also allocated individually or in ranges but, unlike tags, tokens are allocated on a per-connection basis. Since there is no counterpart to the Notifier in NeWS, the application is responsible for the registration of the token in the server as well as keeping track of the wire/token association. Tokens, as provided by the Wire Service, are implemented by using the usertokens provided by CPS. To provide compatibility with applications written using the usertokens, a range of tokens can be reserved and removed from the allocator.

Server-client synchronization

The final component of the Wire Service is the server-client synchronization mechanism. CPS provides a mechanism for a client process to block pending notification from a server process. The Wire Service provides a complementary mechanism which will allow a server process to block pending notification from a client process. This provides symmetrical facilities for synchronous communications.

Notification

The notifier in the wire service reads a single tag from one or more server connections. Depending on its value, the tag is then dispatched to a user defined procedure. Before the message is dispatched, the notifier will set the PostScript and PostScriptInput pointers to the appropriate connection.

`wire_RegisterTag` associates a pointer to a function and a pointer to client data with a specific tag. When the tag reaches the head of the input queue, the

notifier calls the function. The client function associated with a tag can be retrieved using `wire_TagFunction`. The client data is retrieved using `wire_TagData`.

Two modes of notification are offered. For event-driven applications, `wire_EnterNotifer` will allow the notifier to control the dispatch of tags until `wire_ExitNotifier` is called. In this model, the application can eliminate control loops related to tag processing. On the other hand, applications which are not event-driven might need to maintain finer control of message processing. Using `wire_Notify`, these programs can implement a control loop and still use the notifier to dispatch messages.

The `wire_WouldNotify` function reports whether there are pending tags on the input queue of a particular wire or all wires. `wire_SkipEvent` consumes the initial tokens on the wire until the next tag is detected. When no input is pending, this function will not block.

Example

file demo.c:

```
#include            <wire.h>
#include "demo_cps.h"
#define DATA "data"
int                 MouseMoveTag;
int                 MouseDownTag;
int                 MouseExitTag;
int                 *MouseTags[] = {&MouseMoveTag,&MouseDownTag, &MouseExitTag, 0 };
void                MouseMove();
void                MouseDown();
void                MouseExit();

main(argc, argv)
int argc;
char *argv[];
{
        wire_Wire                   thisWire;
                if      ((thisWire = wire_Open(NULL)) == wire_INVALID_WIRE) {
                wire_Perror(argv[0]);
                exit(-1);
        }
        wire_AllocateNamedTags(MouseTags);
```

(continued on next page)

```
        wire_RegisterTag(MouseMoveTag, MouseMove, NULL);
        wire_RegisterTag(MouseDownTag, MouseDown, DATA);
        wire_RegisterTag(MouseExitTag, MouseExit, NULL);
        cps_RegisterTags(MouseMoveTag, MouseDownTag, MouseExitTag);
        cps_MouseMoveTag();
        wire_Notify((struct timeval *)0);
        cps_MouseMoveTag();
        cps_MouseDownTag();
        cps_MouseExitTag();
        wire_EnterNotifier();
        wire_Close(thisWire);
}

void
MouseMove(tag, data)
int                                     tag;
caddr_t                         data;
{

        printf("MouseMove\n");
        wire_SkipEvent();
}

void
MouseDown(tag, data)
int                                     tag;
caddr_t                         data;
{
        printf("MouseDown\n");
        printf("Data = %s\n", data);
        wire_SkipEvent();

}

void
MouseExit(tag,data)
int                                     tag;
caddr_t                         data;
{
        printf("MouseExit\n");
        wire_ExitNotifier();
}
```

file demo_cps.cps:

```
cdef                    cps_RegisterTags(MouseMoveTag, MouseDownTag, MouseExitTag)
```

(continued on next page)

```
        /MoveTag MouseMoveTag def
        /DownTag MouseDownTag def
        /ExitTag MouseExitTag def

cdef cps_MouseMoveTag()
        MoveTag tagprint
cdef cps_MouseDownTag()
        DownTag tagprint
cdef cps_MouseExitTag()
        ExitTag tagprint
```

Connection Management

The client connection to one or more servers is established using the wire service function `wire_Open`. One parameter is passed to `wire_Open` to indicate the server. This parameter can take three forms:

- A null value implying the default server named in either the `NEWSSERVER` or `DISPLAY` environmental variable.
- A parameter in the format "hostname", meaning the default server on the specified host.
- A parameter in the format of either the `NEWSSERVER` string or `DISPLAY` string (see the *X11/NeWS Server Guide*) to direct the connection to the appropriate host.

When the connection to the server is successful, a `wire_Connection` structure is allocated in the wire connection table. This is returned as a `wire_Wire` and is used as a handle to refer to the connection in subsequent calls.

During the open process, the connection is enabled through a call to the `wire_Enable`. One parameter, a `wire_Wire`, is passed to `wire_Enable` to indicate the appropriate connection to activate. In addition, another function, `wire_Disable` is available to disable a connection should the application need to ignore input from the server for a limited time. To disable a wire, the call `wire_Disable(theWire)` is made.

The CPS functions and the server communicate using the PSFILE pointers, PostScript and PostScriptInput. Therefore, the wire service introduces the concept of the "Current Wire." Whenever a `wire_Wire` becomes the current wire, its file pointers are moved into the global variables, PostScript and PostScriptInput. Thus all CPS communication takes place on the Current Wire To make a wire the "Current WIre", a call to `wire_SetCurrent(theWire)` is made and becomes the current wire. The current wire is returned by a call to `wire_Current()`.

Each connection has three function pointers for handling abnormal conditions: death, disease, and unknowntag.

When the connection is terminated abnormally (not via a call to `wire_Close()`), the death function is called.

When the first token in the input queue is not a tag, the notifier calls the disease function. This function is responsible for consuming the offending tokens.

When the first token in the input queue is not a *registered* tag, the notifier calls the unknowntag function.

`wire_Open` provides the new `wire_Wire` with default functions for these problems. The `wire_Problems` function can be used to register private handlers. This function requires four parameters: a `wire_Wire`, a death function pointer, a disease function pointer, and a unknowntag function pointer. A null function pointer value will leave the current function unchanged.

The `wire_Close` function terminates a connection to the server. The `wire_ALLWIRES` constant can be passed to `wire_Enable`, `wire_Disable`, `wire_WouldNotify`, and `wire_Close` to effect all connections. For example, the call, `wire_Close(wire_ALLWIRES)`, would terminate all connections.

An application might need to maintain state information. The `wire_SetData` function can associate a client data pointer with a connection. The information is retrieved using `wire_Data`.

The application can determine the current wire using the `wire_Current` function. The `wire_Valid` function will indicate whether a particular wire is operational.

Example

```
#include <wire.h>
main(argc, argv)
int  argc;
char *argv[];
{
        char                    *client;
        char                    *state = "State Information";
        wire_Wire       thisWire;
        wire_Wire       thatWire;
        if      ((thisWire = wire_Open(NULL)) == wire_INVALID_WIRE) {
                wire_Perror(argv[0]);
                exit(-1);
        }
        thatWire = wire_Current();
        if (thisWire == thatWire)
                printf("The new wire is the current wire\n");
        if      (wire_Valid(thisWire))
                printf("The new wire is operational\n");
        if      (wire_SetData(thisWire, state))
                printf("The new wire has client data : %s\n", state);

        if ((client = wire_Data(thisWire)) && (strcmp(client, state) == 0))
                printf("The client data has been retrieved\n");

        wire_Disable(thisWire);
        if (!(wire_Enabled(thisWire))) {
                printf("The new wire was disabled\n");
                wire_Enable(thisWire);
        }

        wire_Close(thisWire);
        if (!(wire_Valid(thisWire)))
            printf("The new wire has been closed and is no longer operational\n");
}
```

Resource Allocation

Tag Allocation

Using the CPS libraries, the programmer is responsible for the allocation of tags. Without careful supervision, applications may duplicate tag values, leading to confusion and madness. The wire service provides routines to allocate or reserve a range of tags. Tags are no longer constant values, but are generated in a dynamic manner. In addition, tags can be assigned to a list of client variables.

`wire_AllocateTags` reserves a range of client tags. This function is passed an integer (N), which indicates the number of tags required. It returns an integer (M), ensuring that the tag values, (M) through (M+N-1), have not been allocated in the past.

`wire_ReserveTags` allows the wire service to coexist with both CPS and private tag allocation schemes. This function is passed an integer (N), which represents the highest tag value to be reserved. `wire_ReserveTags` should be executed prior to `wire_Open` and `wire_AllocateTags`.

`wire_AllocateNamedTags` assigns tag values to client variables. This function is passed a null terminated array of pointers to integers. It reserves and assigns a tag value to each of the pointers.

Example

```
#include        <wire.h>
#define                 CONSTANT_TAGS 100
int                     MouseMoveTag;
int                     MouseDownTag;
int                     *MouseTags[] = { &MouseMoveTag, &MouseDownTag, 0 };
main(argc, argv)
int argc;
char *argv[];
{
        wire_Wire                       thisWire;
        wire_ReserveTags(CONSTANT_TAGS);
        if ((thisWire = wire_Open(NULL)) == wire_INVALID_WIRE) {
                wire_Perror(argv[0]);
                exit(-1);
        }
        wire_AllocateNamedTags(MouseTags);
        printf("MouseMoveTag = %d\n", MouseMoveTag);
        printf("MouseDownTag = %d\n", MouseDownTag);
        wire_Close(thisWire);
}
```

Token Allocation

The Wire Service references server objects through user defined tokens (*usertoken* as defined in CPS). `wire_AllocateTokens`, `wire_ReserveTokens`, and `wire_AllocateNamedTokens` are similar to the tag allocation functions, with some exceptions. For instance, tokens are allocated on a per connection basis, either tokens must be used with the connection they were defined on or care must be taken to align tokens across connections via the use of `wire_ReserveTokens`. In addition, the application is responsible for registering the usertokens in the server.

Example

file demo.c:

```
#include        <wire.h\>
#include        "demo_cps.h"

int                     MyWindowToken;
int                     MyButtonToken;
int                     *MyTokens[] = { &MyWindowToken, &MyButtonToken, 0 };

main(argc, argv)
int argc;
char *argv[];
{
        wire_Wire               thisWire;
        if      ((thisWire = wire_Open(NULL)) == wire_INVALID_WIRE) {
                wire_Perror(argv[0]);
                exit(-1);
        }
        wire_AllocateNamedTokens(thisWire, MyTokens);
        /* if the same tokens are to be used on another connection,
         * otherWire, the following line would be used. Note that
         * MyButtonToken is the last in the MyTokens array.
         *
         * wire_ReserveTokens(otherWire, MyButtonToken);
         */
        ps_CreateMyWindow(MyWindowToken);
        ps_MapWindow(MyWindowToken);

        wire_Close(thisWire);
}
```

file demo_cps.cps:

```
cdef ps_CreateMyWindow(int token)
        ... /new MyWindowClass send     % MyWindow instance on stack
        token setfileinputtoken         % Register the usertoken/Window pair

cdef ps_MapWindow(token MyWindow)
        /map MyWindow send              % MyWindow gets looked up automatically
```

Server-Client synchronization

The synchronization mechanism provided by the wire service allows a server process to initiate a synchronous call to the client. A call is made to the PostScript function `wire_Synch` with an executable procedure on the stack. The procedure is executed and then `wire_Synch` sends a marker to the client. The server process then blocks until the client acknowledges receipt of the marker.

One of the more common uses of this facility will be to paint of canvases that need some functionality in C for repainting. The server, upon getting the PaintCanvas request, initiates the client code and then blocks before painting the rest of the canvas. In the example below, the client will handle the `REDRAW_TAG` before acknowledging receipt of the marker. This would allow the client to paint the rest of the canvas, if desired.

`wire_Synch` expects to find the *userdict* on the dictionary stack. If `wire_Synch` is to called from another process, make sure that the necessary information is in the current *userdict*.

Example

```
/PaintCanvas { % - => -
      {
         FillColor fillcanvas       % Clear the canvas
         /REDRAW_TAG tagprint       % Pass a tag to client to signal a redraw
         WindowID typedprint        % Pass token to identify which window
         flush                      % Flush the buffers
      } wire_Sync                   % Block pending client notification.
} def
```

Building a typical application

A typical application consists of three or more files containing the C routines, the code to be loaded into the server, and the cps code.

`main.c`: This file would contain the C code for the application. It should include the file `<wire.h>` and `"main_cps.h"` and be linked with libwire.a and libcps.a.

`main_cps.cps`: This file should contain the CPS code for communicating to the server. It should be run through the CPS preprocessor to generate `main_cps.h` which is included in main.c

`main.ps`: This contains most of the PostScript code needed by the application. It is typically loaded through a CPS call defined in `main_cps.cps`.

Note: Assuming that the environment variable `$OPENWINHOME` is set to the directory containing a properly installed version of OpenWindows, then the following consists of a sample compilation:

```
$ $OPENWINHOME/bin/cps main_cps.cps
$ cc -I$OPENWINHOME/include main.c $OPENWINHOME/lib/libcps.a $OPENWINHOME/lib/libwire.a
$
```

3. CANVASES

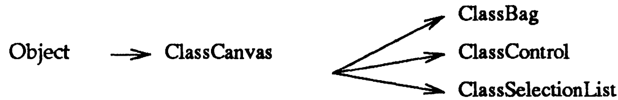
3. CANVASES

3 Canvases

Canvases 3-1
Introduction 3-1
Canvas Creation & Destruction 3-2
Canvas Appearance 3-3
Activation and Deactivation 3-4
Canvas Damage Repair 3-5
Help and Menus 3-6
Canvas Tree Manipulation and Enumeration 3-6
Canvas Geometry 3-8
User Interaction Utilities 3-9
Canvas Validation 3-10
Canvas Cursors 3-11
Canvas Focus Management 3-11

Canvases

Introduction



The NeWS canvas is a surface on which the PostScript language can be used to perform drawing operations. Canvases are arranged in a hierarchical manner, with the root being the device canvas or framebuffer. Both the shape and location of a canvas can be altered. While the default shape is rectangular, the NeWS operator **reshapecanvas** will change the shape to match the region outlined by the current path.

The NeWS Development Environment's ClassCanvas combines a NeWS canvas object and an event manager process to create a self-sufficient user interface item. Examples of ClassCanvas subclasses span the range of simple user interface objects like buttons, sliders and scrollbars to more elaborate objects such as windows and menus. Even complete applications such as word processors and drawing editors are defined as a text canvas or a drawing canvas augmented by associated menus and property sheets.

ClassCanvas itself is rather minimal, assuming that most interesting user interface objects will be created as subclasses. Clients creating such objects will therefore need to know the details of creating and designing a subclass of ClassCanvas.

ClassCanvas instances are "smart NeWS canvases", being able to manage many of the attributes of their NeWS canvas object. Thus they can paint themselves, repair their damaged areas, determine their shape, and map and unmap themselves. In the following discussion, "canvas" is used to mean an instance of the Toolkit's ClassCanvas; using "NeWS canvas" to refer to the NeWS canvas object. Although is it useful to make this distinction in the following discussion, the NeWS canvas object is actually the same object as the ClassCanvas instance; this being done by adding the necessary keys to the NeWS canvas object to turn it into an instance.

Canvases can access their parent, sibling, and child canvases; providing the basis for a "container hierarchy". One particular ClassCanvas subclass, ClassBag, relies on this capability to implement compound canvas instances such as windows, property sheets and command frames, all of which are composed of nested canvases. (See Chapter 4, "Managing Groups of Canvases" for information on ClassBag.)

Certain attributes of canvases inherit through this container hierarchy rather than through the class hierarchy. For example, the FillColor, StrokeColor, and TextColor of canvas instances default to being their parent canvas'.

A canvas can be either opaque or transparent. Drawing occurs on an opaque canvas that will also conceal the regions of canvases beneath it. You can draw to transparent canvas but anything drawn on a transparent canvas is painted on the first opaque canvas beneath it. A transparent canvas is used to define regions which are sensitive to input but do not interfere with drawing in other canvases.

Canvas Creation & Destruction

Canvases are created by calling **/new** with the parent canvas (either ClassCanvas instance or NeWS canvas object) as an argument. The canvas will be a created as a child of the given parent canvas. The initial values of several of the corresponding NeWS canvas attributes may be specified as class variables. Thus redefining the class variable /Transparent or /Mapped will change the initial value of the corresponding NeWS canvas.

When a canvas is destroyed, it deactivates itself and tells the system to remove any references to it. References to the canvas that are not known by the canvas are presumed to be "soft" or under client control. The client must remove any reference to the canvas to complete the garbage collection of the canvas. The most obvious reference is the client handle to the canvas created during **/new**. Less obvious are references left on the operand or dict stacks. Similarly, the canvas cannot be the current canvas for any process. Reference "leaks" will be quite obvious: the canvas will not go away from the screen. (Note : **/destroy** does *not* unmap the canvas.)

Methods:
/new
/newinit
/destroy
/Retained
/SaveBehind
/Transparent
/Mapped
/EventsConsumed

Canvas Appearance

Canvases are responsible for their shape and graphical contents. The graphical contents of a canvas is determined by the **/PaintCanvas** method which defaults to using the values of several ClassCanvas class variables. The canvas is filled with FillColor, and the edge of the canvas is stroked with StrokeColor using a stroke width of BorderStroke. The colors may be manipulated with **/setcolors** and **/getcolors**.

A TextColor and TextFont are also available for canvases containing text. The font is initialized from the parameters TextFamily, TextSize, and TextEncoding. The font may be manipulated by **/settextparams, /textparams, /settextfont** and **/textfont**.

The **/PaintCanvas** method may be set using **/setpaintproc**. The procedure argument will be method compiled, thus can contain "self" and "super" usage. Four utilities are provided for filling and stroking the canvas by the painting procedure.

/scroll provides a simple painting aid for scrolling. This procedure simply uses the NeWS operator **copyarea** to displace the current contents of the canvas the desired amount. **/scroll** then calls **/PaintScrolledArea** which is generally overridden to be "smart." The default sets the clip path properly (even for oddly shaped canvases) and calls **/PaintCanvas.**

Subclassers may override the default rectangular shape by overriding the **/path** method. This method simply takes a bounding box and creates a path just fitting that box.

Methods:	
/paint	/PaintCanvas
/setpaintproc	/FillCanvas
/setcolors	/StrokeCanvas
/getcolors	/StrokeAndFillCanvas
/settextparams	/FillCanvasInterior
/textparams	/FillColor
/settextfont	/StrokeColor
/textfont	/BorderStroke
/scroll	/TextColor
/path	/TextFamily
	/TextSize
	/TextEncoding
	/TextFont
	/PaintScrolledArea

Activation and Deactivation

Each canvas has an associated event manager that may be activated and deactivated. Damage is handled by this event manager, as are the help facility and automatic menu management.

When activated, a canvas uses its parent canvas' event manager if there is one, otherwise it creates its own. The canvas creates the interests it wants the event manager to manage in the **/MakeInterests** method (see chapter 7, "Interests"). These interests must contain their own callback; see the section "Executable Matches" in Chapter 7, "Interests."

Methods:	
/activate	/MakeInterests
/deactivate	/CreateEventMgr
/active?	
/eventmgr	

Canvas Damage Repair

When the Retained attribute is true, the window system is requested to store a duplicate of the canvas pixels in memory. As the canvas suffers damage, the window system repairs the damage using the copy. The use of Retained is a hint which means that your **/PaintCanvas** or **/FixCanvas** method is not called.

When opaque canvases are mapped, or when part of a canvas is exposed by moving a covering canvas, the canvas is sent a /Damaged event. The ClassCanvas event manager responds to this event by calling the canvas' **/HandleFix** method. This method sets the canvas clip path, then calls the **/fix** method. (Note: the canvas clip is not the same as the clip path.)

The **/fix** method checks to see whether the intersection of the canvasclip and the canvas is empty. If not, it calls the **/FixCanvas** method which defaults to **/PaintCanvas**. The test is made as a trivial rejection test to avoid unnecessary painting.

Although the transparency of a canvas is set during initialization via the class variable /Transparent, it can also be dynamically changed using the **/settransparent** method.

A sub-canvas of the opaque canvas handling damage may also desire forked painting. For example, in response to a menu command, the "client" transparent canvas of an application may have to repaint itself. It may call **/damage** on itself; which will cause the event manager of the opaque parent to fork damage repair.

When the SaveBehind attribute is true (the default for Toolkit menus), the window system is requested to save pixels underneath this canvas when it is displayed. When this canvas is no longer displayed, the server should be able to repair the damage using the cached pixels. The SaveBehind attribute is useful for transitory canvases such as menus. It is also a hint.

Methods:	
/fix	**/Transparent**
/settransparent	**/HandleFix**
/transparent?	**/FixCanvas**

Help and Menus

The default canvas event management also handles triggering a help procedure from the HelpKey, and popping up a menu from the MenuButton.

When the HelpKey is pressed (as determined by the current UI), the HelpProc is called if non-null. This may be dynamically set using the **/sethelp** method, or can be set by the subclass via the /HelpProc class variable. Clients may cause the help proc to be executed directly through some alternative user interface by simply calling **/callhelp**. **/help** returns the current HelpProc.

Similarly, if the MenuButton is pressed, the CanvasMenu is popped up and activated if non-null. The menu may be provided either by subclassers via the CanvasMenu class variable, or by clients via the **/setmenu** method. Menu call-backs generally will require the ClassTarget facility. If AutoTargetMenu is true, the canvas is installed as the menu's target just prior to popping up.

Both help and menu may be set even after the canvas is activated; it will modify the running event manager properly. The same holds for damage discussed in the preceeding section.

Methods:	
/sethelp	**/HelpProc**
/help	**/CanvasMenu**
/callhelp	**/AutoTargetMenu**
/setmenu	
/menu	
/autotargetmenu	
/autotargetmenu?	

Canvas Tree Manipulation and Enumeration

The NeWS canvases are maintained in a tree rooted at the framebuffer. Because the NeWS canvas objects are identical to the ClassCanvas instances, the instances are automatically organized into a tree. (For an explanation of the difference between the canvas tree and the class tree see the *NeWS Programmer's Guide.*) The initial position of a canvas in the canvas tree is determined by the parent argument to /new. This can be changed using the **/reparent** method.

The canvas can change its location among its siblings using **/totop** and **/tobottom.**

There are many enumeration procedures for retrieving canvases relative to the current canvas. **/parent** returns the parent canvas while **/parents** returns all parents to the framebuffer. **/parentdescendant** returns the first parent descending from a given class between the given canvas and the framebuffer. **/parentdescendant** is useful, for example, for a canvas to get its window frame:

```
mycanvas /parentdescendant ClassFrame send
```

All of a canvas' siblings, including itself, are obtained via **/siblings+** and **/siblings-**; the "+" version being from back to front. A list of a canvas' siblings above or below but, not including itself, is obtained by using **/siblingsabove** or **/siblingsbelow**. A list of all my children is obtained by **/children+** or **/children-**; the "+" & "-" as with siblings. /descendants returns all canvases below me, including me.

For performance reasons, these lists do not check that the canvases are true instances. To remove non-instances from an array, use /FilterNonInstances.

The canvas tree is used to implement a container hierarchy. An object is said to inherit through the container hierarchy if it includes a class variable that defaults to its parent's value. Thus the colors all inherit though the container hierarchy. This is logical: it makes more sense for a color to default to its parent's value than its superclasses.

Methods:	
/new	**/siblings-**
/reparent	**/siblingsabove**
/totop	**/siblingsbelow**
/tobottom	**/children+**
/parent	**/children-**
/parents	**/descendants**
/parentdescendant	**/FilterNonInstances**
/siblings+	

Canvas Geometry

The canvas may be reshaped or moved to change its size or location. Reshaping is done by specifying a bounding box; the canvas will reshape itself, using **/path**, to just fill that bbox. You move a canvas by specifying the position of the lower left corner of its bounding box. Both operations specify their coordinates using the CTM (current transform matrix). This means that whatever scale, rotation, and translation is in effect will be used during the reshape or move.

This use of the CTM is vital to the flexibility of the NeWS Development Environment. It allows a vertical scrollbar to be made from a horizontal scrollbar by simply rotating the CTM before placing it. It allows scaling of an entire application by reshaping its window with the CTM scaled. It allows nesting of windows within other windows because they do not assume they are positioned relative to the framebuffer.

Each NeWS canvas object has its own coordinate system; that which is in effect when "reshapecanvs" is called. This defines the CTM established each time "setcanvas" is called. The NeWS Development Environment has adopted the protocol that both **/reshape** and **/move** specify their coordinates relative to the bounding box with orgin at the lower left corner. The canvas itself may use any scale, and even translation it likes. It must, however, return the /size, /location, and /bbox in the caller's CTM with a lower-left origin.

To help with separating the caller's coordinates and the canvas' coordinates, the Transform utility tranaslates from the callers space to the canvas'. This defaults to a simple translate to the caller's lower left corner. It is commonly overriden to give the canvas a 0-1 coordinate system to simplify calculations:

```
/Transform { % x y w h => 0 0 1 1
        4 2 roll          % w h x y
        translate scale           % -
        0 0 1 1
} def
```

In sum, the caller "owns" the positioning and size of the canvas in whatever CTM desired; the canvas itself "owns" its default (private) CTM.

Canvases typically manage data which imposes size constraints on the canvas. The canvas should not be sized below its **/minsize**. The canvas might also have a **/preferredsize** somewhat greater than that size. Thus a text editor might choose to display at least one line of 20 characters as its minsize, but choose a preferred size of 80 characters by 40 lines. The **/reshape** method does not inforce the minsize limit. This must be done by the client. This is done to avoid unnecessary checking that the client can more easily do. Typically the client will "know" enough of the semantics of the application canvas layout to impose these limits in a simple manner. **/lockminsize** may be used by a client to install a simple, constant, non-calculated, minsize in a canvas after performing the calculation the first time.

Methods:	
/reshape	**/Transform**
/move	**/lockminsize**
/size	**/preferredsize**
/location	
/bbox	
/minsize	

User Interaction Utilities

ClassCanvas provides simple user interaction methods with reasonable default bahavior. These procedures are based on the "getfromuser" utility (triggered on UpTransition) and use the /path method to provide reasonable default behavior.

The utilities are typically called after a DownTransition has been processed, often in an interest provided in the canvas' MakeInterests. The following would provide trivial dynamic resizing and moving of the associated canvas:

```
/MakeInterests { % - => interestlist
        /MakeInterests super send
        /LeftMouseButton {pop /movefromuser}
        /DownTransition Canvas MakeInterest
        /MiddleMouseButton /stretchcorner
        /DownTransition Canvas MakeInterest
} def
```

Methods:
/stretchcorner
/bboxfromuser
/reshapefromuser
/movefromuser

Canvas Validation

To help optimize flicker-free, high performance re-painting of canvases after changes to their contents, ClassCanvas supports a simple validation scheme. When changes are made to a canvas that cause its image to need updating, the canvas is sent **/invalidate**. When the canvas next is told to paint itself, it first checks to see if it is valid. If not, it sends itself the **/validate** method. Note that several invalidations may be made without causing the repaint to occur.

/validate should be overridden to change any internal parameters of the canvas required by the PaintCanvas routine. ClassBag, for example, will call its **/layout** method from **/validate.**

Methods:
/invalidate
/validate
/valid?
/?validate

Canvas Cursors

ClassCanvas supports defaulting the cursor associated with a canvas, and setting this cursor interactively. This facility attempts to use shared "well known" cursors; cursors whose name is known to the system. Generally subclassers simply define the class variable /CursorImage to be a well known cursor name. The CursorMask is assumed to be the next glyph in the cursor font; set it to a non-null value to override this assumption. Use the **/setcursor** method to use any non-shared cursor you might need. The names of the NeWS Development Environment's cursors are:

/basic	/panning	/rtarr
/move	/navigation	/xhair
/copy	/nouse	/xcurs
/busy	/ptr	/hourg
/stop	/beye	

Methods:	Subclassing Methods:
/setcursor	**/CursorImage** **/CursorMask**

Canvas Focus Management

For an explanation of Focus Management see Chapter 4, Managing Groups of Canvases, section "Focus Management."

4. MANAGING GROUPS OF CANVASES

4. MANAGING GROUPS OF CANVASES

4 Managing Groups of Canvases

Managing Groups of Canvases 4-1
Introduction 4-1
Bags 4-1
- Creation and destruction 4-2
- Insertions and Removals 4-3
- Access to Bag Clients 4-5
- Graphics State Utilities 4-6
- Sizing Protocols 4-6
- Layout and Invalidation 4-7
- Activation and Event Management 4-8
- Painting and Damage Repair 4-9

Containers 4-10
- Client Naming 4-10
- Creating a Container 4-10
- Getting and Setting the Client 4-11
- Size Negotiations 4-12

OpenLookPane 4-13
- Controlling the Scrollbars 4-13

Frames 4-14
- Frame Attributes 4-15
- Opening, Closing and Zooming 4-16
- Manipulating a Frame Menu's Default Behavior 4-17
- Subframes 4-18
- Notification 4-19
- Selection and Focus 4-20
- Freezing 4-20
- /demo Method 4-21
- Frame Class Hierarchy 4-21
- OPEN LOOK Frames 4-22
- Frame Size and Placement 4-23
- Subframe Functions 4-23
- Shared Frame Menus 4-24
- Instantiating Frames 4-24
- Subclassing Frames 4-24
- Adding Frame Attributes 4-26

Utility Bags 4-28
- AbsoluteBag 4-29
- RowColumnBags 4-31
- Flex Bags 4-32

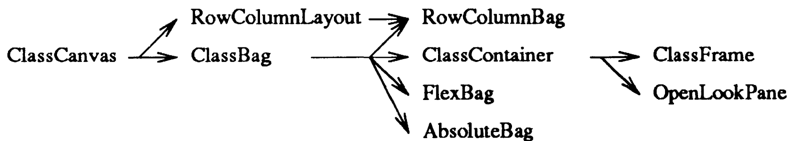
Example of Bag Usage and Subclassing 4-35
Focus Management 4-39
- Focus Definitions 4-40
- Focus Forwarding 4-41
- Focus Noticing 4-41
- How Focus Forwarding and Noticing Works 4-42

Managing Groups of Canvases

Introduction

A bag is a canvas that is designed explicitly to manage a group of 'child' NeWS Development Environment canvases. (See the *NeWS Programmer's Guide* an explanation of the parent/child relationships in the NeWS canvas hierarchy.) ClassBag is the basis for many of the NeWS Development Environment's components — frames are bags, panes are bags, an application's control area will typically be a bag of control canvases. Whenever a group of canvases are brought together to achieve some effect, a bag will manage them.

Figure 4-1: Bags



Bags

As an intrinsic class ClassBag is rarely instantiated directly. It is designed expressly for subclassing. ClassBag extends the NeWS canvas hierarchy model in the following important ways:

- Child canvases (called 'clients' of the bag) can be given names when they are added to the bag. These names may be used later to refer to these canvases. This feature obviates the need for application programs to hold references to every canvas in their application.
- Instances of any subclass of ClassGraphic (commonly referred to as 'graphics') can also be managed as clients of the bag.
- Bags explicitly manage the layout of their clients. Different bags have different layout policies, but ClassBag controls when the layout procedure is activated. Because the NeWS Development Environment assumes that layout may be an expensive operation, bags minimize the number of times the layout procedure is called by the use of a validation scheme.

- Bags manage the sharing of event managers. In a typical toolkit application there is one event manager process watching for mouse actions and keystrokes for each frame in the application. Via its activation and deactivation primitives, ClassBag allows every canvas in a frame to share the same event manager.
- Bags manage the damage repair for their transparent child canvases. This effectively means that most canvases inside a frame can be transparent, and a significant space improvement can be gained over using opaque canvases. The bag will also take care of painting the graphics they manage when damage occurs. This allows graphics to be treated as lightweight canvases which are not sensitive to input.

Creation and destruction

Like all canvases, bags expect a canvas object to be on the top of the stack when **/new** is called. This canvas becomes the parent of the newly created canvas. There is no requirement that the parent of a bag be another bag.

When **/destroy** is sent to a bag it also sends **/destroydependent** to each of its clients. This defaults to sending **/destroy** to them. Whole bag hierarchies can be destroyed in this way. Frames for example will destroy all their ornaments and their client subtree when **/destroy** is sent to them.

If you wish to prevent some client of a bag from being destroyed when the bag is destroyed, you should override the client's **/destroydependent** method, and not allow it to default to **/destroy**. Canvases that are still bag clients after **/destroydependent** is called are reparented to an offscreen canvas so that the bag can be disposed of completely.

Methods:
/new **/newinit** **/destroy** **/destroydependent**

Insertions and Removals

The **/addclient** method is used to insert a canvas or a graphic into a bag. (See section 2 on ClassBag for a detailed explanation of the syntax of **/addclient**.) There are three independent choices to make when adding a client to a bag.

1. The client may be an instance or a class. If the client is ClassCanvas or *any* subclass of ClassCanvas then the class itself may be used as an argument to **/addclient**. For other classes (e.g., ClassGraphic) **/addclient** must be given an instance. Given a class, the bag will send **/new** to the class to obtain the instance. Adding clients as classes is slightly faster (NeWS does not have to reparent the client canvas) and more compact than preinstantiating them. If a class is supplied, then the application must also provide arguments for the later instantiation of the class. For example, the above bag insertion could be done using the class OpenLookButton in the following way:

```
b1 [20 50 (Foo) {button-callback} OpenLookButton] /addclient mybag send
```

   The button's label and callback are the arguments that will be handed to **/new** by the bag in this example. (A parent canvas argument is *not* provided with the class because the bag itself will become the parent of the newly created canvas.) The **/NewClient** method is used to instantiate clients when a class is added. **/NewClient** defaults to calling **/newdefault** on the class.

2. A client may or may not be named. If a name is provided for **/addclient** the client can be retrieved later by providing the same name to **/getbyname**. If 'null' is given as a name for all bag clients, then bags save space by not maintaining the name/client mapping. Access the clients by insertion order in this case. For example, '0 /getbyname bag send' returns the first client inserted in the bag.

3. A client may or may not have "baggage." Baggage is unformatted information used by subclasses of ClassBag. Typically it provides layout information to the subclass. For example, an AbsoluteBag requires that two numbers (the x,y coordinates) be provided as baggage when adding a client. These numbers get wrapped in an array along with the instance:

```
/b1 [20 50 mybutton] /addclient mybag send
```

When adding a client to a subclass that does not require baggage (for example RowColumnBag), the array is not required:

```
/b1 mybutton /addclient mybag send
```

Sophisticated subclasses that wish to change the way client references are maintained by the bag may override /RegisterClient and /DeRegister-Client. For example, if an application is keeping references to every client of the bag via some external mechanism the application can prevent the bag from duplicating this effort by overriding these methods and providing its own **/clientlist** method.

To remove a client from a bag use **/removeclient**. This method expects an integer (if the client was added with a null name), or the client itself as arguments. To move an object from one bag to another it must first be removed from the first, and then added to the second.

Methods:
/addclient
/removeclient
/getbyname
/NewClient
/RegisterClient
/DeRegisterClient

Access to Bag Clients

There are a number of methods for interacting with the clients of a bag.

/baggage and **/setbaggage** can be used to retrieve and modify the subclass-specific information passed in during **/addclient**. Bag layout procedures for example, usually call **/baggage** on each client to get their layout information.

The number of clients is returned by **/clientcount**. The set of clients is returned as an array by **/clientlist**. **/graphicclientcount** and **/graphicclientlist** provide the same information, but only for those clients that are graphics. To retrieve the list of clients that are canvases, use the ClassCanvas methods **/children+** or **/children-**.

An arbitrary object can be presented to **/client?**. A boolean is returned; it indicates whether or not this object is a client of the bag. Note that the bag does not maintain a reverse index from objects to names. Given an object that is known to be in the bag there is no way to find out what name was specified for it during **/addclient**.

A convenience wrapper around **/clientlist** is **/foreachclient**. It takes a procedure and **foralls** it over the list of clients. **/sendclient** sends a method to a named client of the bag. For example, to change the label on the button in the previous example call:

```
(Bar) /setlabel /b1 /sendclient bag send
```

Methods:
/baggage
/setbaggage
/client?
/clientcount
/clientlist
/graphicclientcount
/graphicclientlist
/children+
/children-
/foreachclient
/sendclient

Graphics State Utilities

When subclassing a bag it is often desirable to establish the bag's canvas as the current canvas. This is especially true when measuring the size of a bag client, or laying the bag out. ClassBag provides two utilities, **/BagBegin** and **/BagEnd** which it uses internally to set and restore the graphics state for this purpose. Certain subclasser methods are called from within a **/BagBegin /BagEnd** context, and it is important to take advantage of this so that the subclasser doesn't duplicate the effort. The methods **/Layout, /Minsize, /PreferredSize** are currently treated this way. Each will be discussed below.

Methods:
/BagBegin
/BagEnd

Sizing Protocols

ClassCanvas establishes two important sizing interfaces: **/minsize** and **/preferredsize.** ClassBag turns these into protocols by making them hierarchical: most bags when sent **/minsize** ask their clients the same question, and use the results to calculate their own answer. **/preferredsize** works in the same way.

ClassBag cannot perform this recursive **/minsize** (or **/preferredsize**) calculation itself. It does not know how to combine the various answers it would get. Subclasses of ClassBag do have this knowledge however, and are expected to take their clients into account when calculating **/minsize** or **/preferredsize.** For

example, the **/minsize** for ClassContainer asks **/minsize** of its main client, then adds a pair of constants to leave room for the borders of the container.

If the minimum size calculation for a subclass of ClassBag does not require that the bag be established as the current canvas, then you can override **/minsize** directly. If an application requires that the bag be the current canvas it should override **/MinSize**. In **/MinSize** one can assume that the graphics state has been set up correctly. The same applies for **/preferredsize** and **/PreferredSize**.

Methods:
/minsize
/MinSize
/preferredsize
/PreferredSize

Layout and Invalidation

ClassBag uses the term 'layout' to mean the positioning (and perhaps resizing) of bag clients once the size and shape of the bag itself has been established. Performing layout, by overriding the **/Layout** method, is the responsibility of all subclasses of ClassBag. The bag will be the current canvas when **/Layout** is called, so it is the job of the subclasser to **/move** or **/reshape** each client of the bag to take advantage of the current **/size** of the bag.

Applications do not normally call **/layout** (which calls **/Layout** inside a **/BagBegin, /BagEnd** pair) themselves. Instead, each bag maintains a 'valid' flag which it uses to decide whether layout is necessary. This flag is checked as the first phase of both **/paint** and **/fix**. If the layout has been invalidated, **/layout** is called before painting begins in earnest. The methods which automatically invalidate the layout are: **/reshape, /addclient,** and **/removeclient**. An application that does *not* wish to have the layout invalidated when any of these methods are called, should override **/?invalidate.** To manually invalidate the layout (perhaps because the geometry of one of the clients changed in some way), applications should call the bag's **/invalidate** method.

Most applications that use bags will never call **/layout, /validate, /?validate, /invalidate, /?invalidate,** or **/valid?**. They will simply override **/Layout** and rely on the validation protocol to call this method when layout is required.

Methods:
/layout
/layout
/?validate
/?invalidate
/invalidate
/valid?
/validate
/reshape

Activation and Event Management

ClassBag plays an important role in the event management for a hierarchy of canvases. The ClassCanvas **/activate** is responsible for searching up the canvas hierarchy looking for an already active canvas (one that has an event manager process handling it). If it fails to find an event manager **/activate** will create one. After an event manager is found or created, the canvas's **/MakeInterests** method is called and the resulting interests are expressed in the event manager process.

ClassBag overrides **/activate** and augments the ClassCanvas' **/activate** by recursively calling **/activate** *down* the tree on its client canvases. The upshot of this is that a single **/activate** method sent to the root of a bag hierarchy causes every *canvas* in the tree to become active. All their interests are expressed in a single event manager process created at the root of the tree. In particular, to make every canvas inside an application's frame sensitive to input, it suffices to send **/activate** to the frame.

/deactivate undoes the effect of **/activate.** It removes the interests for every canvas in the bag hierarchy rooted at the canvas to which the method was sent. It then kills the event manager process if this process was created expressly for the given bag.

/active? returns true if there is an event manager process alive and fielding events on this canvas.

Removing a client from the bag (via **/removeclient**) causes it to be deactivated if and only if it shares an event manager with the bag. In practice this means that if an application activated a client before putting it into the bag the client will still be active after removing it from the bag.

Methods:
/activate
/deactivate
/active?

Painting and Damage Repair

ClassCanvas establishes simple interpretations for **/paint** and **/fix**. **/paint** is called by the application to manually refresh a canvas. When NeWS generates a /Damage event **/fix** is called to repaint some damaged portion of a canvas.

ClassBag extends these methods by making them recursive. The **/paint** method sent to a bag not only paints the bag's canvas (by calling its **/PaintCanvas** method), but also calls **/paint** on its mapped child canvases, using **/PaintChildren.** A bag's **/paint** method also has the responsibility for painting the graphic clients of the bag and does so by calling **/PaintGraphicChildren.** The recursive nature of ClassBag's **/paint** is responsible for the fact that a **/paint** sent to a frame will refresh every canvas inside the frame.

The **/fix** protocol is a little more complex because NeWS distributes **/Damage** events only to the opaque canvases in the damaged area. The **/fix** method in ClassBag only takes responsibility for recursively fixing those child canvases that are transparent. **/FixChildren** propagates **/fix** down the tree, and **/PaintGraphicChildren** is still called on each bag encountered in the recursion. The end result is that **/FixCanvas** is called once for every canvas inside or overlapping the damaged region.

Methods:	
/paint	**/FixChildren**
/fix	**/PaintChildren**
/FixCanvas	**/PaintGraphicChildren**
/PaintCanvas	

Containers

A container is a kind of bag that is specialized to handle one major client, and zero or more minor ones. Usually the minor clients are allocated a fixed amount of real estate in the bag, and the major one grows and shrinks as the bag changes size.

ClassContainer is not designed to be instantiated directly. It is subclassed within the toolkit, and these subclasses are themselves instantiated.

Frames are the most important subclass of containers. The major client of a frame is the canvas that occupies the interior of the window. The minor clients are the ornaments (resize corners, close box, footer, etc.) that surround the window interior. When a frame is reshaped the minor clients are moved to new positions on the edge of the frame, and the major client is reshaped to take up all the remaining area.

Because of its special significance the major client of a container is often referred to simply as 'the client' of the container. Hence we often refer to the interior canvas of a frame as 'the client of the frame'.

Client Naming

ClassBag leaves the business of associating a name with each client up to the application. ClassContainer restricts client naming in the following important way: the major client of a container always has the name '/Client'. This is in fact how ClassContainer distinguishes the major client from the others. When adding minor clients to a container, you may give them any name other than /Client.

Creating a Container

Containers expect the major client to be presented as an argument to **/new** when creating an instance of some subclass of ClassContainer. In common with ClassBag's **/addclient** syntax, this client argument may be either an instance, or an array containing a class and the arguments required to instantiate that class. For example, there are two ways to create a frame containing a vanilla ClassCanvas instance:

```
framebuffer /new ClassCanvas send
[] framebuffer /new OpenLookBaseFrame send
```

in which an instance of the canvas is handed in, or

```
[ClassCanvas] [] framebuffer /new OpenLookBaseFrame send
```

in which a class is handed in, and the container automatically instantiates this class. The second version above is not only more terse, it is also more efficient. The client canvas does not need to be reparented from the framebuffer. It is created with the container as its parent.

It is also legal to present 'null' as the client argument when instantiating a container. Until set otherwise such a container will have no client whatsoever.

Methods:
/new /newinit

Getting and Setting the Client

The major client of a container can be changed at any time by calling **/setclient** on that container. **/setclient** returns the previous client (if any) of the container.

This current client is returned by the method **/client**. (Note that **/client** is just a thinly veiled call to **/getbyname** using the /Client name.)

Methods:
/setclient
/client

Size Negotiations

Containers maintain four class variables to help position the major client inside the container. These are the (usually constant) amounts of border space to leave to the top, right, bottom and left of the major client when laying it out. The method /BorderWidths sums up the left and right borders, and /BorderHeights sums the top and bottom space.

/fitclient takes a proposed client size, and returns the total size of the container necessary to give the client the specified width and height. **/unfitclient** does the reverse. It tells you how big the client will become if the container is reshaped to the given size. Both these methods make use of /BorderWidths and /BorderHeights.

The **/minsize** of a container is the **/minsize** of the major client added onto the size of the borders. Similarly, the /preferredsize of the container is the /preferredsize of the client plus the size of the borders.

Methods:	
/fitclient	**/BorderLeft**
/unfitclient	**/BorderBottom**
	/BorderRight
	/BorderTop
	/BorderHeights
	/BorderWidths

OpenLookPane

OpenLookPane is a subclass of ClassContainer whose job is arrange either one or two scrollbars around some canvas. This canvas is the major client (named /Client) of the container, and will presumably be manipulated by the scrollbars.

The scrollbars are not automatically connected to the client canvas. To use an OpenLookPane you must not only provide a client canvas, but also provide the callbacks for the scrollbar(s) to update it. (See Chapter 6, Controls, the ScrollBar section.)

Controlling the Scrollbars

The class variables **/UseHSbar?** and **/UseVSbar?** control whether or not the pane, when instantiated, will have a horizontal and/or a vertical scrollbar. Vertical scrollbars appear to the right of the client canvas, horizontal ones to the bottom.

The subclasser methods **/CreateVerticalScrollbar** and **/CreateHorizontalScrollbar** by default return instances of OpenLookVerticalScrollbar and OpenLookHorizontalScrollbar respectively. You should override these if you have your own scrollbar that you wish the pane to use.

The subclasser methods **/CreateHSbarNotify** and **/CreateVSbarNotify** return the notification procedures for the scrollbars. Override these methods to make your scrollbars control your client canvas.

Methods:
/CreateVerticalScrollbar **/CreateHorizontalScrollbar** **/CreateHSbarNotify** **/CreateVSbarNotify**

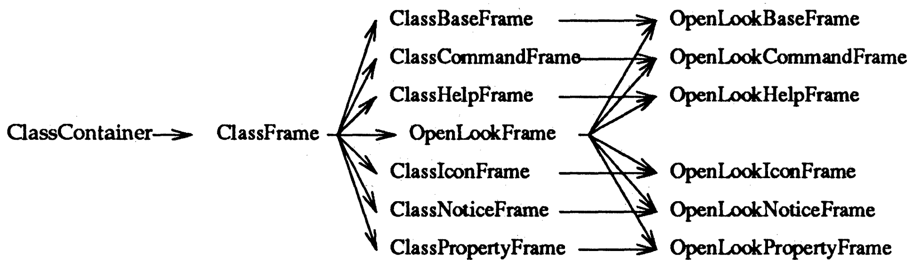
Class Variables:
UseHSbar? **/UseVSbar?**

Pane Sizing

The **/minsize** of a pane is the maximum of the **/minsize** of the client canvas and the minsize of whichever scrollbars happen to be present. Currently there is no analagous calculation for the **/preferredsize** of an OpenLookPane. The **/preferredsize** thus defaults to the pane's /minsize.

Methods:
/minsize

Figure 4-2: Frame Hierarchy



Frames

ClassFrame implements window frames. It is an intrinsic class that provides a set of protocols and utility methods that are useful for many different types of windows. As an intrinsic class, ClassFrame is not intended to be instantiated itself. Instead it provides a framework on which to build subclasses that can be instantiated. OPEN LOOK frames are the classes meant to be instantiated.

A frame manages a single "client" canvas. The frame wraps that canvas with a border that may include various ornaments managed by the frame. Examples of ornaments are a title area, reshape controls, close or grow boxes, etc.

A frame has a number of attributes, e.g. "is it reshapeable?", or "can it be closed?" There will be some visual indication of these attributes, often in the form of some ornament, e.g. a control or graphic. Or, there may be a menu item corresponding to the attribute. ClassFrame defines a minimal protocol for these attributes, but leaves the implementation of ornaments or menu choices to subclasses. The frame is a bag, so the subclass may use it to hold any ornaments.

A frame can have associated with it some number of secondary frames, known as "subframes". ClassFrame defines an icon to be a standard subframe, but leaves the creation of an icon to subclasses. Other subframes may be added by subclasses. ClassFrame provides mechanisms for managing the list of subframes and ways to send messages between superframes and subframes.

Other intrinsic frame facilities include the ability to be selected and to have the input focus, notification of the client when the frame's state changes as the result of a user interaction, temporarily freezing processing of events, opening and closing (i.e. unmapping a frame and mapping its associated icon frame), zooming the frame to a larger size, and restoring ("unzooming") it to its normal size.

Frame Attributes

A frame has a set of attributes, each having a name and a boolean value. The standard set of attributes for ClassFrame are:

/Close	True to allow closing the frame to an icon
/Footer	True to display a footer area
/Label	True to display a label (title) area
/Pin	True to allow the frame to be pinned (stay up)
/Reshape	True to allow the frame to be reshaped

The attributes usually correspond to some sort of control, e.g., a close box or label (title) area. The attributes are known outside the frame by their names, e.g., /Label. Each attribute has a default value specified by the frame subclass. These default values may be overridden for a particular frame instance with a parameter to **/new** when the frame is created. You may query or change the attributes for an existing frame with the **/frameattribute** and **/setframeattribute** methods.

The implementation and behavior of frame attributes is entirely up to frame subclasses. A client user of a frame should make no assumptions about how the attributes are displayed to the user or how they are implemented. A typical implementation by a subclass is to create a control corresponding to an attribute and to put the control in the frame's bag. Or, a subclass may use an attribute to enable or disable an item on the frame's menu. Another possibility is to use the attribute simply to enable or disable some feature of the frame with no visible indication, e.g., the /Reshape attribute could simply allow a frame to be reshaped when the user presses a mouse button on part of its border.

Each of the standard attributes has several related methods. For example, the /Footer attribute has methods **/setfooter** and **/footer** to set and retrieve the footer messages for the frame.

Additional attributes may be defined by subclasses.

Methods:
/new
/frameattribute
/setframeattribute
/footer
/setfooter
/label
/setlabel

Opening, Closing and Zooming

An intrinsic notion of frames is that they may be closed to an icon, zoomed to a larger size, and opened or unzoomed back to their normal size. An icon is a subframe with the predefined name /Icon. When a frame is closed, it is unmapped and its icon subframe, if present, is mapped. Icons may be any size or shape, and may display any image. When a frame is zoomed, it is reshaped to a larger size. By default, the larger size is the width of the unzoomed window and the height of the framebuffer.

Methods:
/open
/opened?
/flipiconic
/zoom
/zoomed?
/flipzoom

Manipulating a Frame Menu's Default Behavior

You can intercept common user-issued window commands such as refresh, zoom, close (iconicize), reshape and quit. The next five subsections explain the methods you can use to change the toolkit's default actions for these commands.

The client canvas of a frame will be stretched to take up the available space *automatically* when the frame is reshaped. If this is not the behavior you desire you can override the client's **/reshape** method.

In order to constrain the way in which a frame can be reshaped, several options are available. One way to prevent reshaping altogether for a time is by dynamically adding and removing the resize-corners via the **/setframeattribute** method (see below). To prevent a frame from being made smaller than some size, specify a minsize for your client canvas (or the frame itself). Even more complex nondefault reshape behavior is possible by overriding the frame's **/reshape** method. For example if you wanted a frame with dimensions that were always some multiple of 10 points the override to **/reshape** would be:

```
/reshape { % x y w h -> -
        10 div round 10 mul exch
        10 div round 10 mul exch
        /reshape super send
} def
```

This would change the width and height arguments before executing the default reshape action.

Refresh

Some applications may wish to be informed that "refresh" was called from the frame menu (as opposed to a normal damage event). For example, calling refresh from the window menu might cause the client to reprocess some data file rather than just initiate a repaint. This behavior can be achieved by overriding the **/paint** method. When a frame receives explicit refreshes (like those called from the frame menu) **/paint** is called. Thus to change the behavior of explicit refreshes **/paint** must be overridden.

Iconicize

Some users may want to know when a window is iconicized. For example, a game may want to suspend the game clock until the window is reopened. To achieve this and similar behavior, override **/open**. **/open** takes a boolean argument, false for iconifying the frame, true for deiconifying it.

Zoom

Override **/zoom** to implement changes to a frame's behavior when Zoom is selected from the frame menu. Like **/open**, **/zoom** takes a boolean argument.

Quit

You can interpose on "Quit" by overriding **/destroyfromuser** in the frame. This, in turn, overrides the quit sent from both the frame and icon menus. For example, if an application wanted to put up a confirming notice before the quit was executed **/destroyfromuser** would be overridden to show it. The callback from the confirming button would send the destroy method to the frame.

Subframes

A frame (the "superframe") may manage one or more subframes. ClassFrame defines the following behavior for subframes:

- Destroying a superframe destroys its subframes.
- Subframes share the event manager of their superframe.
- Activating/deactivating a superframe does the same for its subframes.
- Freezing/unfreezing a superframe does the same for its subframes.

A frame has a dictionary of its subframes, so each subframe has a name associated with it. The subframe list is managed with the **/addsubframe** and **/removesubframe** methods. Each subframe has a reference to its superframe, accessed with the **/superframe** method. Subframes may be nested more than one level deep.

Methods:
/addsubframe
/removesubframe
/subframe
/subframe?
/subframes
/superframe
/rootframe
/sendsubframe
/sendsuperframe

Notification

Frames provide a notification mechanism similar to that of controls. The client application may provide a notification procedure, using **/setnotifyproc**, that will be called whenever the state of the frame changes as the result of a user interaction. The notify proc can obtain the reason for the notification via the **/notifyreason** method.

Currently only a small number of interactions cause notification to happen. These are:

Reason	Interaction
/NotifyPin	Frame pinned/unpinned
/NotifyProps	Property sheet brought up
/NotifyReset	Reset chosen in the property frame menu
/NotifyApply	Apply chosen in property frame menu

The above four names are the values obtained from a call to the **/notifyreason** method from the frame's notify proc. The notify proc is set via **/setnotifyproc**. This simply means that some event occurred, e.g., a frame was unpinned. You can do something when you are notified that the event occurred or you can ignore it.

Methods:
/setnotifyproc
/notifyproc
/setnotifyreason
/notifyreason
/callnotify

Selection and Focus

A frame may be selected or given the input focus. Typically this will be done automatically by selection and focus managers. Frames have methods to give visual feedback that they are selected or have the focus.

Methods:
/reflectfocus
/reflectselected
/setfocus
/focus?
/setselected
/selected?
/selectedframes
/notifyselected
/sendselected

Freezing

A frame may be "frozen", i.e., made to ignore most events. Typically a frame is frozen because the user is being notified of a situation such as an error and must give some input before the application can proceed. When a frame is frozen, the only events it processes are those for damage repair and loss of focus or selection. A frozen frame is not the same as the *OPEN LOOK UI Specification* defines as busy.

Methods:
/freeze
/freezeall

/demo Method

ClassFrame includes a **/demo** method that can be sent to any derived subclass. The method sends **/new** to the class to which **/demo** was sent. Examples:

```
/demo OpenLookHelpFrame send
/demo OpenLookBaseFrame send
```

Frame Class Hierarchy

There are six intrinsic frame types, defined in subclasses of ClassFrame:

ClassBaseFrame
ClassCommandFrame
ClassHelpFrame
ClassIconFrame
ClassNoticeFrame
ClassPropertyFrame

These classes do not provide any additional functionality beyond that provided by ClassFrame. These classes exist to be abstract superclasses that have corresponding look and feel classes, such as OpenLookBaseFrame. These classes will typically be used as follows. By a client:

```
<parameters> /new OpenLookBaseFrame
```

By a subclasser:

```
/MyBaseFrame /defaultclass ClassBaseFrame send []
classbegin
  ...
classend def
```

ClassFrame contains behavior that is shared by all frame types. The frame type classes contain behavior that is particular to the specific types. You should subclass ClassFrame to define data and methods that are shared by more than one frame type. You should subclass the frame type classes, mixing in your subclass of ClassFrame, to define the individual frame types. In many cases a particular behavior will be shared by most frame types but for one frame type it will be different. Use the class hierarchy to implement this - define the common behavior in your ClassFrame subclass, then override the appropriate methods in the class for the different frame type. This is preferable to having a single shared method that has conditional code based on frame type.

OPEN LOOK Frames

Class OpenLookFrame is a subclass of ClassFrame that implements functionality shared by OPEN LOOK frame types. The six OPEN LOOK frame types are implemented as subclasses of the six intrinsic frame types. Multiple inheritance is used to give each of these classes two superclasses: the intrinsic frame class and class OpenLookFrame. OpenLookFrame implements appearance and behavior that is shared by all the OPEN LOOK frame types. The class, and several helper classes, implements the frame label, footer, reshape corners, menu button, etc. These ornaments correspond to the frame attributes, e.g., /Reshape, /Footer, defined in ClassFrame. OpenLookFrame also includes the following features that are shared among more than one frame type including OpenLook-BaseFrame. Behavior that is particular to one frame type is typically implemented in that type's subclass.

Frame Size and Placement

Like all canvases, frames have the notion of a preferred size. By default a frame's preferred size is a frame large enough to display its owner, label, and footer in their entirety. Frames also display their clients at the clients preferred size. To change this default, override **/preferredsize**.

The **/place** method computes a default size and placement for a frame. If the frame has already has been reshaped that size is preserved and the frame is only moved in response to a place message. If the frame has a superframe (see below for an explanation of superframes) it is positioned so its upper left corner is coincident with the upper left corner of the super frame. If the frame doesn't have a superframe (as most base frames don't) then the frame is positioned successively down the diagonal of the screen starting at the upper left corner.

If the frame has not been shaped yet (i.e., it has no size), **/place** shapes it to its preferred size. It is then positioned either relative to its superframe or to a default location based on the gravity setting. In ClassBaseFrame and its superclasses **/place** defaults to **/reshapefromuser.**

The gravity setting is used to calculate default positions for frames. For OPEN LOOK frames the gravity setting also determines where **/place** will start tiling frames. Frame gravity is set by sending **/setgravity** to a specific frame class. The choices for gravity setting are dependent upon the particular frame class: normal OPEN LOOK frames expect names like /UpperLeft, /UpperRight, etc. Icons expect names like /Top, /Bottom, etc. For OpenLookBaseFrames gravity defaults to /UpperLeft. **/seticongravity** is supported for backwards compatibility. It just calls **/setgravity** on the base frame's icon.

Methods:
/place **/setgravity**

Subframe Functions

OpenLookFrame overrides the **/open** method so when a superframe is opened or closed all its subframes are also opened or closed. The **/toptop** and **/tobottom** methods are also overridden so subframes are sent to the top or bottom with their superframe.

Methods:
/open /opensubframes /closesubframes /totop /tobottom

Shared Frame Menus

OpenLookFrame creates menus that are shared by all frames. There is one menu that is shared between base and icon frames, and another that is shared between property and command frames. If a client needs to modify the menu for a particular frame, the code that creates the frame menu should be copied and modified. This code can be found in OLframe.ps. Once modified the new menu should be stored with **/setmenu**.

Instantiating Frames

The frame subclasses whose names begin with "OpenLook" are the subclasses that you should instantiate. Each frame type subclass implements behavior that is particular to that type of frame. For example, OpenLookBaseFrame automatically creates an icon frame as a subframe when a base frame is created. Another example is the override of the **/open** method in class OpenLookHelpFrame. It calls the **/pin** method whenever a help frame is opened, since all OPEN LOOK help frames are supposed to be pinned when they are opened.

Subclassing Frames

This section describes techniques for defining your own frame subclasses.

Subclassing a Single Frame Type

The following is sample code to subclass a single frame type:

```
/MyBaseFrame /defaultclass ClassBaseFrame send []
classbegin
classend def
```

You should generally send **/defaultclass** to the intrinsic frame class instead of subclassing one of the OPEN LOOK frame classes directly. If your subclass builds on the variables and methods in the intrinsic class rather than the Open Look class, it should be possible for someone to change the default look and feel and still use your subclass.

Subclassing Several Frame Types

Here's some standard code to subclass more than one frame type:

```
% Class containing things that are shared
%  among frame types.
%
/MyFrame /defaultclass ClassFrame send []
classbegin
        % Shared frame functionality
        /BaseFrameClass {MyBaseFrame} def
        /IconFrameClass {MyIconFrame} def
classend def

% Individual frame types

/MyBaseFrame [MyFrame /defaultclass ClassBaseFrame send] []
classbegin
        % Functionality specific to base frames
classend def

/MyIconFrame [MyFrame /defaultclass ClassIconFrame send] []
classbegin
        % Functionality specific to icon frames
classend def
```

Class MyFrame contains everything that is needed by more than one frame type. Things needed by a single frame type are defined in that type's class.

Each of the frame type classes has two superclasses: MyFrame, providing common new functionality; and the default implementation of the intrinsic frame class, providing the default frame type's functionality.

ClassFrame contains utility methods (for example, **/BaseFrameCreate**) that can be used by subclasses to create frames that are associated with one another. For example, OPEN LOOK base frames have associated icon frames, so OpenLook-BaseFrame calls **/IconFrameCreate**. There are two levels of control that subclassers can use to change the way associated frames are created. First, the /FooFrameCreate methods can be overridden. This provides the most flexibility, but is often more than is needed. If the only thing a subclass wants to do is to change the class that is instantiated, it can override the /FooFrameClass methods to return the proper classes. Each /FooFrameCreate method in ClassFrame instantiates the class returned by /FooFrameClass. This is done in the example above, where /BaseFrameClass and /IconFrameClass are overridden to return MyBaseFrame and MyIconFrame. The default frame classes (e.g., OpenLookHelpFrame) are used for the other frame types. Note that the /FooFrameClass's are executable procedures rather than direct references to the classes. Deferring the evaluation of the classes avoids the problem of the classes not being defined yet when class MyFrame is being defined.

Adding Frame Attributes

Adding a new frame attribute in a subclass is simple. Here's an outline of how you might add a new control to the frame border. For this example, let's assume the attribute controls whether the frame is "zoomable", that is whether it can grow to the full height of the screen. If it is zoomable, the frame will have a "zoom box" control in its border.

First, define the new attribute by defining a variable in the subclass. The boolean value of the class variable is the default value used by all instances of this class. This value can be overridden for a particular instance by parameters to the **/new** or **/setframeattribute** methods.

```
/Zoom    true    def        % True to allow frame to be zoomed
```

Override the /Ornaments method to include the new attribute. This method returns attribute names on the stack and is used to construct an array of attributes that have associated ornaments, such as controls or graphics. The ornaments are created and laid out in the order they are returned by the **/Ornaments** method. If the painting order of the new ornament is important, the override method should do a super send then search through the attributes on the stack and insert the new one in the appropriate place. If the painting order doesn't matter, the method can simply add the new ornament at the end of the list:

```
% Override - add /Zoom ornament
/Ornaments { % - => name1 ... nameN
        /Ornaments super send
        /Zoom
} def
```

The subclass must provide the following methods for the new control:

```
% Create the Zoom ornament.
%
/ZoomCreate { % - => -
        % Typically will do something like this:
        /Zoom [<args> class] /addclient self send
} def

% Destroy the Zoom ornament.
%
/ZoomDestroy { % - => -
        % Typically something like:
        /Zoom /deletebyname self send {/destroy exch send} if
} def

% Lay out the Zoom ornament.
%
/ZoomLayout { % - => -
        % Typically something like:
        /Zoom /getbyname self send {
                % x y width height calculations
                /reshape 6 -1 roll send
        } if
} def

% Notify method for the Zoom ornament.
%
/ZoomNotify { % zoom-control => -
        pop
        /zoomed? self send not /zoom self send
} def
```

Utility Bags

There are many ways an application might want to layout the clients of a bag. The NeWS Development Environment includes three utility bags that provide support for laying out an arbitrary number of arbitrarily-sized bag clients in three different ways. The names of the subclasses are: AbsoluteBag, RowColumnBag and FlexBag.

AbsoluteBag

The absolute in the name of this bag refers to the location of the AbsoluteBag's clients. AbsoluteBags position their clients at application-specified x,y coordinates and keep them there no matter what the size of the bag. Clients added to AbsoluteBags must have an x,y coordinate as baggage. (For more information on baggage see the "Insertion and Removals" section above.)

minsize

When the minsize message is sent to AbsoluteBag it attempts to calculate a reasonable size for itself based on the x,y coordinates of the clients and their sizes (*not* their minsizes). Basically, AbsoluteBag uses the position and size information of its clients to calculate the smallest bounding box that fits all its clients. Thus AbsoluteBag's minsize calculation tries to ensure that all its clients are visible. However, the minsized layout many not be "pretty."

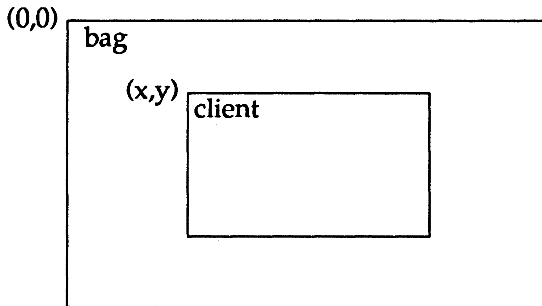
If a class is passed to **/addclient** then AbsoluteBag creates a minsized client.

Coordinate system

Only absolute bags support two orientations of the coordinate system: top-down (the default) where the origin is at the upper left corner of the bag; and the normal NeWS bottom-up system where the origin is at the lower left corner of the bag. Use **/settopdown** to change the orientation of the origin in absolute bags; **/settopdown** takes a boolean argument, true for top-down and false for bottom-up.

In the top-down system the coordinates given to the bag for each client are taken as the distance from the top-left corner of the bag to the top-left corner of the client :

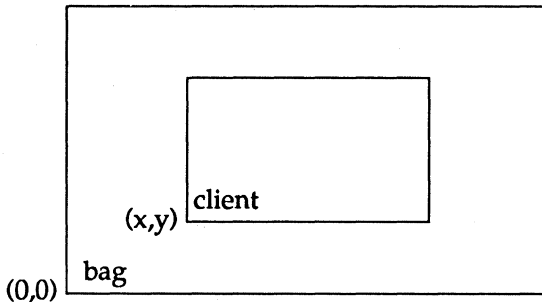
Figure 4-3: Top Down Coordinates



true /topdown

(absolute bags only)

Figure 4-4: Bottom Up Coordinates



false /topdown

(standard NeWS coordinates)

RowColumnBags

As the name of this ClassBag subclass implies RowColumnBags are designed to lay out clients in a grid of rows and columns. Clients added to RowColumnBags do *not* take any baggage. The default layout for RowColumnBags is row major order, with one column and as many rows as there are clients. This layout is identical to the default layout of menus. RowColumnBags inherit their layout from the RowColumnLayout mixin class.

The arrangement of RowColumnBag clients can be changed using **/setlayoutstyle**. **/setlayoutstyle** takes three arguments: a boolean first argument to determine row or column major layout (true for row major; false for column major), the number of rows and the number of columns. If the number of rows and columns are specified then those numbers of rows and columns are created. If one of the row/column arguments is null then the appropriate value is calculated by dividing the number of items to be displayed by the other, known value. If null is specified for both arguments then the bag is layed out in the default style.

Every cell in the grid of a RowColumnBag is the same size. When calculating its minsize RowColumnBag uses the maximum minsize of all its client's minsizes. Thus if a RowColumnBag had six clients of varying minsizes, it would determine which of the six clients had the largest minsize and multiply the size by six (the number of clients) in order to calculate its own minsize.

By default RowColumnBags do not put any space between cells of the layout grid or between the grid and the borders of the bag. To change the default spacing use **/setgaps** to add horizontal and/or vertical spacing between cells. **/setgaps** takes two arguments the horizontal gap (or null) and the vertical gap (or null). The spacing is given in points. Use **/gaps** to determine the amount of space between cells. **/gaps** returns two numbers, the horizontal gap and the vertical gap. Both values are in points.

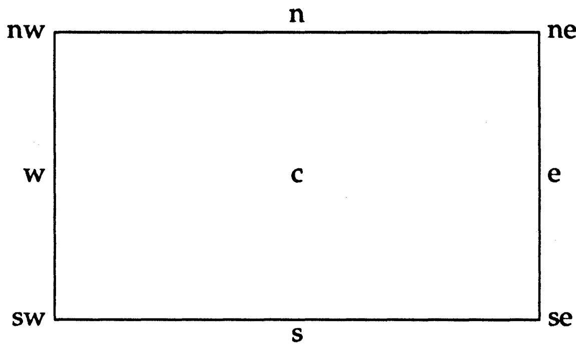
You can also change the size of the border by using **/setborder**. **/setborder** takes the number of points of white space you want between the client grid and the bag's inside edge.

Flex Bags

A FlexBag is a bag in which the positions of clients are determined by executable code they pass in during **/addclient**. This code is executed each time the bag is layed out. Compass-point notation is used so that clients may be placed relative to a corner of another client. To aid you in relative positioning of clients, utilities are provided. See the section "FlexBag Positioning Utilities" below. FlexBag use the NeWS bottom-up coordinate system.

The compass-point notation refers to a client's bounding box (c is center):

Figure 4-5: Compass Point Notation



Adding Clients

The order of insertion into a FlexBag is critical if the bag is being used for relative layout. During each call to **/Layout** the position code is executed in the order in which the clients were inserted. In other words, do not make the position of an earlier addition depend on the position of a later one. Incorrect results will follow.

FlexBag's layout code expects clients to be passed to **/addclient** using the following form:

```
/name [compass-point {executable} client] /addclient aflexbag send
```

The executable must return x,y coordinates that determine where the given compass-point of the given client is placed in the bag. The client can be either a class (if it is a canvas) or an instance. For example:

```
/myclient [/sw {200 300} mybutton]
/addclient myflexbag send
```

puts the bottom left corner of mybutton at position 200 300 in myflexbag. Similarly:

```
/myclient [/sw {200 300}
(Button1) nullproc OpenLookButton]
/addclient myflexbag send
```

would create an instance of OpenLookButton with the label Button1 and no callback, make it a client of myflexbag, and place the bottom left corner of the button at position 200 300 in myflexbag.

FlexBag also recognizes the compass-point notation in reference to itself. When positioning clients relative to the bag's coordinates the FlexBag code recognizes the following executable form:

```
{bag-compass-point self position-proc}
```

Thus:

```
/c {/c self POSITION 10 10 XYADD}
```

places the client so that its center will be offset 10,10 from the center of the bag. See the section "FlexBag Position Utilities" below for an explanation of POSITION and XYADD.

Clients can be added to flex bags with no layout information. In this case you should set the bag's default layout specification by using **/setlayoutspec**. **/setlayoutspec** takes the same arguments as you use when adding clients, a compass-point (or null) and an executable that returns a position.

minsize

FlexBags calculate their minsize based on the positioning information given to the bag when clients are added. You should note that while FlexBag don't presume that any shape or size client is being put in them the FlexBag code attempts to make an "intelligent" guess as to what its minsize should be by using a heuristic. For complicated relational positioning of FlexBag clients the heuristic may yield an arrangement of the clients that you find unacceptable. If that should occur use the ClassCanvas method **/lockminsize** to override FlexBag's calculation.

FlexBag Positioning Utilities

FlexBags provide five positioning utilities. They are /POSITION, /WIDTH, /HEIGHT, /XYADD, and /XYSUB. These utilities are especially useful for positioning clients relative either to other clients or to the flex bag itself. POSITION, WIDTH, and HEIGHT take either a canvas instance, the name of a client or "Previous" or "Current" which refer to the previous and current clients, respectively.

/POSITION takes a compass-point and a client's name, a canvas, or a bag and returns the x,y position of the compass-point of that canvas. For example, to position a client so that it is always in the center of the bag an application could do:

```
[/c      {/c      self     POSITION} . . .]
 |        |        |
client   bag      bag
center   center
```

This code fragment makes the client's center and the bag's center coincide.

/WIDTH takes a client name, a bag, or a canvas and returns its width in points.

/HEIGHT takes a client name, a bag, or a canvas and returns its height in points.

An application could use /WIDTH and /HEIGHT together to position a client relative to the size of the bag:

```
[/c {self WIDTH .25 mul self HEIGHT .5 mul} . . .]
```

This example positions the center of a client 1/4 of the flex bag's width and at 1/2 the bag's height. Since these are relative positions they are preserved independently of the bag's size.

/XYADD does 2-D vector addition on two sets of x,y coordinates. The syntax is: x1 y1 x2 y2 **XYADD** x1+x2 y1 +y2.

/XYSUB does 2-D vector subtraction on two sets of x,y coordinates. The syntax is: x1 y1 x2 y2 **XYSUB** x2-x1 y2 -y1.

Example of Bag Usage and Subclassing

The following example code shows how an application programmer might typically subclass ClassBag, and use the result as the client of an OPEN LOOK frame (or any bag for that matter).

```
% The task of a SimpleAppBag is to provide the layout policy for a basic
% OpenLook application. The inside of the window in such an application
% is occupied by two areas — a control area of fixed height, and a
% stretchable canvas that is the main focus of interraction. A
% SimpleAppBag manages these two canvases.
% The control area sits above the stretchable canvas, and both must
% adjust their widths to fill the frame interior when the user reshapes
% the frame.

/SimpleAppBag ClassBag []
classbegin
        % This bag always has exactly two clients. It expects them
        % to be presented as arguments to /new, and hence consumes them
        % during /newinit. This bag takes the responsibility of giving
        % names to clients: /Fixed for the upper area, and /Floating
        % for the lower stretchable region. No baggage is required for
        % clients of a SimpleAppBag.
        %
        /newinit { % fixed-client floating-client -> -
                /newinit super send
                /Floating exch /addclient self send
                /Fixed exch /addclient self send
                {
                        % We set the height of the /Fixed client once to its
                        % ideal size, and never alter in again. Arguably, we
                        % shouldn't mess with it at all and just trust the
                        % application to hand in a canvas with the desired height.
                        % This approach fails though when the application passes in a
                        % class rather than an instance.
                        %
                        % The "0 0" and width arguments to /reshape are just
                        % placeholders — they are overridden in
                        % /Layout.
                        %
                        0 0 /minsize self send /reshape self send
                } /Fixed /sendclient self send
        } def

        % Here's where we decide how small we allow the user to make
        % this bag and its clients. Since we aren't overriding
        % /preferredsize as well, its /preferrsedsize will default to
        % its /minsize. In the calculation below, we say that the min width
        % is the maximum of the min widths for each of the clients, and the
        % min height is the sum of the min heights of the clients. This is
        % the obvious choice given the layout of the bag.
        %
```

(continued on next page)

```
        % Subtle point: we override /MinSize rather than /minsize
        % just in case one of our clients is a graphic rather than a
        % canvas as expected. When graphics calculate their minimum size
        % they make use of the current canvas. Overriding /MinSize allows
        % us to assume that the bag *is* the current canvas.
        %
        /MinSize { % - -> w h
                /minsize /Fixed /sendclient self send          % w1 h1
                /minsize /Floating /sendclient self send       % w1 h1 w2 h2
                3 -1 roll add 3 1 roll max exch                % max(w1,w2) h1+h2
        } def

        % This procedure is called automatically just before painting
        % the bag *if* something has changed in the bag's geometry since
        % the last time it was called. See Layout and Invalidation'
        %
        % Here we reshape the two clients to fill the space available
        % in the bag (as returned by /size self send). The /Fixed client
        % may be stretched horizontally, but its height will not change.
        % The floating client will be streched in both dimensions to
        % take up the rest of the space.
        %
        /Layout { % - -> -
                /size /Fixed /sendclient self send exch pop    % h-Fi
                /size self send                                % h-Fi w h
                2 index sub                                    % h-Fi w h'
                0 1 index 3 index 6 -1 roll                    % w h' 0 w h-Fi
                /reshape /Fixed /sendclient self send          % w h'
                0 0 4 2 roll /reshape /Floating /sendclient self send
        } def

        % This type of bag doesn't want to stroke its border or fill itself
        % so we give it a null paint proc.
        %
        /PaintCanvas nullproc def
classend def

% Below we show how a SimpleAppBag might be used in an application.
% We make the /Fixed client a FlexBag with a couple of controls in it,
% and the /Floating client a brightly colored canvas.
%

/ColorCanvas ClassCanvas []
classbegin
        /setrgb { % rvalue gvalue bvalue -> -
                /FillColor 4 1 roll rgbcolor def
```

(continued on next page)

```
                /paint self send
        } def

        /minsize { % - -> minw minh
                50 50
        } def
classend def

/ControlArea FlexBag []
classbegin
        /PaintCanvas nullproc def            % No paint is fine here

        % Set every client to its preferred size, when it is added
        % to the bag. This obviates the need for tedious reshaping
        % code outside the bag. /RegisterBag is a sweet spot for getting
        % at every instance inserted in the bag.
        %
        /RegisterClient { % name client -> -
                0 0 /preferredsize 3 index send /reshape 5 index send
                /RegisterClient super send
        } def
classend def

% Make the stretchable color canvas
/cc framebuffer /new ColorCanvas send def

% Make three buttons to put in the flexbag
/bb (Black) {0 0 0 /setrgb /sendtarget 6 -1 roll send}
        framebuffer /new OpenLookButton send def

/wb (White) {1 1 1 /setrgb /sendtarget 6 -1 roll send}
        framebuffer /new OpenLookButton send def

/rb (Random) {random random random /setrgb /sendtarget 6 -1 roll send}
        framebuffer /new OpenLookButton send def

% Make the color canvas the target of all buttons.
cc /settarget bb send
cc /settarget wb send
cc /settarget rb send

% Make the flexbag and add the buttons to it, and add a little
% white space around the buttons.
/fb framebuffer /new ControlArea send def
null [/nw {/nw self POSITION 10 -10 XYADD} bb] /addclient fb send
null [/nw {/sw Previous POSITION 10 sub} wb] /addclient fb send
```

(continued on next page)

```
null [/ne (/ne self POSITION 10 10 XYSUB) rb] /addclient fb send
10 10 /setpadding fb send

% Create a frame containing a SimpleAppBag which in turn contains
% the flexbag and the color canvas.

/win [fb cc SimpleAppBag] [/Footer false]
        framebuffer /new OpenLookBaseFrame send def
(Bag Subclassing) /setlabel win send

/place win send
/activate win send
/map win send

newprocessgroup
currentfile closefile
```

Focus Management

ClassBag provides mechanisms for "focus forwarding" and "focus noticing". When a bag receives the input focus, it may forward the focus to another canvas that is an immediate child or a more remote descendant. A bag may be interested in noticing when the focus is given to one of its descendants directly. These concepts are best explained with an example.

Suppose you have an OPEN LOOK frame (whose class is a subclass of ClassBag) containing a control area (a bag) containing a text control. You want the user to be able to click the mouse on the frame and then to be able to type to the text control. Clicking on the frame will give it the input focus. However, it is not the frame that is interested in key strokes, but rather the text control, so the frame must forward the focus to the text control. The frame title area highlights when the input focus is anywhere inside the frame. This means the frame must notice the change of focus, even if the user clicks the mouse directly on the text control so it gets the focus without any intervention from the frame.

Note: the above example assumes the click-to-type focus style is being used. Focus forwarding and noticing also works with the follow mouse focus style. The mechanism built into ClassBag and used by ClassFrame allow behavior such as that described above to be implemented very simply. How this is done is described below.

Focus Definitions

Focus Client

A canvas interested in receiving the input focus, either for the purpose of procesing keys itself or for passing the focus on to another canvas that will consume keys. A canvas is designated a focus client via the utility functions addfocusclient and removefocusclient.

Key Consumer

A canvas that processes keyboard input. Set by the /setkeyconsumer method in ClassCanvas. A key consumer must also express appropriate interests in keys.

Focus Forwarder

A bag that upon receiving the input focus passes the focus on to another canvas. The automatic mechanism for doing this is controlled by the /FocusForwarder? class variable in ClassBag. Focus forwarding is generally an attribute of an entire class, but may be enabled for specific instances by promoting /FocusForwarder?.

Focus Target

For a focus forwarder bag, the canvas to which the input focus will be forwarded by the automatic forwarding mechanism. The focus target is maintained automatically, but may be set explicitly with the **/setfocustarget** method in ClassBag.

Focus Noticer

A bag that is interested in being notified when the input focus enters or leaves itself or any of its descendant canvases. Controlled by the class variable /FocusNoticer? in ClassBag.

Focus Forwarding

All key consumers are generally focus clients, but not all focus clients are key consumers. A focus client that is not a key consumer is usually a focus forwarder, transferring the input focus to a descendant canvas that is a key consumer.

ClassBag provides a mechanism for automatically handling focus forwarding. In the simplest case, a class that is a subclass of bag simply sets the /FocusForwarder? class variable to true. This causes an appropriate interest to be created by /MakeInterests so the bag will be able to receive the input focus. It also causes FocusTarget to be initialized and maintained for the bag. When the input focus is given to the bag, it is automatically transfered to the focus target. More elaborate behavior can be achieved by overriding methods in a subclass of ClassBag.

Class Variable:
/FocusForwarder?

The variable /FocusForwarder? is defined as false in ClassBag and true in OpenLookFrame. A subclass of ClassBag that is not a subclass of OpenLookFrame must redefine this variable to be true to enable focus forwarding.

Focus Noticing

ClassBag provides a simple mechanism for noticing when the input focus enters or leaves the bag or any of its descendant canvases. In the simplest case, a subclass sets the /FocusNoticer? class variable to true. This causes an appropriate interest to be created by /MakeInterests. The subclass then overrides the /NoticeFocusEnterExit method to take whatever action is necessary, such as providing some sort of highlighting.

Class Variable:
/FocusNoticer?

Method:
/NoticeFocusEnterExit

The variable /FocusNoticer? is defined as false in ClassBag and true in OpenLookFrame. A subclass of ClassBag that is not a subclass of OpenLookFrame must redefine this variable to be true to enable focus noticing.

How Focus Forwarding and Noticing Works

What follows is a detailed description of the mechanisms involved in focus forwarding and noticing. This is useful for subclassers who want to change some aspect of this behavior.

When a focus forwarder bag or any of its descendant canvases gets the input focus, the /NoticeFocus method is called. This method calls /NoticeSelfFocus if the focus is for the bag itself or /NoticeDescendantFocus if the focus is for a descendant canvas. /NoticeSelfFocus checks to see if there is a focus target (i.e. if FocusTarget is not null), and if so transfers the focus to that canvas via the /TransferFocus method. /NoticeDescendantFocus sets FocusTarget to the canvas receiving the focus. This makes sure FocusTarget is set to the most recent focus recipient, even when focus forwarding does not take place, e.g. when the focus goes from outside the bag directly to a key consumer canvas or when the focus goes from one key consumer canvas to another within the same bag.

Methods:
/NoticeFocus **/NoticeSelfFocus** **/NoticeDescendantFocus** **/TransferFocus**

When a new value for FocusTarget is needed, the **/MakeFocusTarget** method is called. For example, it is called when adding a key consumer canvas to a bag that has no focus target, or when removing a key consumer canvas that is the focus target. A subclasser can override **/MakeFocusTarget** to implement algorithms for determining which canvas should be the focus target. By default, the first key consumer canvas added to a bag becomes the initial focus target, and FocusTarget is set to null if the current focus target is removed.

Methods:
/MakeFocusTarget

When a key consumer canvas is added to a bag, the **/addkeyconsumer** method is called for the bag. This method calls /addfocusdescendant for itself and all of the bags from its parent to the framebuffer. The /addfocusdescendant method sets the focus target if it is not already set. The /addfocusdescendant message is sent to all bags up the canvas hierarchy so they can all have the opportunity to set their focus targets. An analogous procedure takes place when a key consumer canvas is removed from a bag. The **/removekeyconsumer** method is called and it calls /removefocusdescendant up the canvas hierarchy. The /removefocusdescendant method calls /MakeFocusTarget to get a new focus target if the canvas being removed is the current focus target for the bag. The /addkeyconsumer and /removekeyconsumer methods are also called when a change is made to whether a canvas is a key consumer.

Methods:
/addkeyconsumer **/removekeyconsumer** **/addfocusdescendant** **/removefocusdescendant** **/setkeyconsumer (ClassCanvas)**

Subclassers way want to develop algorithms for setting a bag's focus target based on the last times canvases had the input focus. For example, a ClassBag subclass might override /MakeFocusTarget so that when the current focus target canvas is removed from a bag other key consumer canvases in the bag are checked and the one that had the focus most recently becomes the new focus target. ClassCanvas provides the **/setlastfocustime** and **/lastfocustime** methods as a standard way of storing and retrieving the last time a canvas had the input focus. Key consumer canvases should call /setlastfocustime when they receive the input focus. This will allow them to be placed in bags that use last focus time.

Methods:
/lastfocustime (ClassCanvas)
/setlastfocustime (ClassCanvas)

Methods:
/setfocustarget
/focustarget
/setkeyconsumer
/removefocusdescendant
/removekeyconsumer
/addfocusdescendant
/addkeyconsumer
/lastfocustime
/MakeFocusTarget
/NoticeDescendantFocus
/NoticeFocus
/NoticeFocusEnterExit
/NoticeSelfFocus

Class Variables:
/FocusForwarder?
/FocusNoticer?
/FocusTarget

5. MENUS AND OTHER SELECTION LISTS

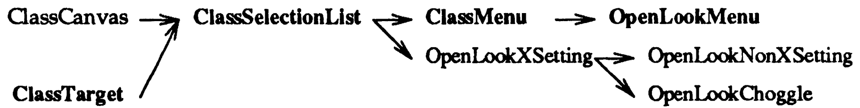
5. MENUS AND OTHER SELECTION LISTS

5 Menus and Other Selection Lists

Menus and Other Selection Lists	5-1
Introduction	5-1
Menus	5-1
■ Introduction	5-1
■ Creating Menus	5-2
■ Laying Out Menus	5-4
■ Manipulating Menus	5-5
■ Menu Values	5-5
■ Pinned Menus	5-6
■ Callbacks, Targets and /setmenu	5-6
Settings	5-8
■ OpenLookXSetting	5-8
■ OpenLookChoggle	5-12
OpenLookNonXSetting	5-12
■ Example	5-13

Menus and Other Selection Lists

Introduction



ClassSelectionList is the basis for menus and setting controls (Exclusive, NonExclusive and Choggles). It is not designed to be instantiated directly. The idea behind a selection list is that a single canvas manages a grid of regularly spaced items that can be independently selected via the mouse. The most common use of selection lists is for menus.

Menus

This section covers the more important details of the programmer's interface for OPEN LOOK menus. For information on using OPEN LOOK menus please consult the *OPEN LOOK UI Style Guide*.

This section concentrates on the OpenLookMenu class. This section covers those methods deemed most important for using menus. The complete set of methods associated with OpenLookMenu and its superclasses can be found in section 2.

Introduction

ClassMenu is an intrinsic class implementing menus; it supports hierarchical pop-up pinnable menus. ClassMenu, as with other intrinsic classes, is meant to be subclassed rather than instantiated; it provides the foundation for OpenLookMenu. You can subclass ClassMenu if you want functionality that is substantially different from that which OPEN LOOK menus provides. And while OpenLookMenu is generally intended for instantiation, it can be subclassed to implement small changes in the class, such as having all menus come up using a font that differs from the default.

Applications can automatically associate a menu with another canvas. The NeWS Development Environment takes special care to manage the relationship between canvases and their associated menus. By default, an instance of ClassCanvas does not have a menu but ClassCanvas has been designed to

expect a menu. The NeWS Development Environment provides procedures to facilitate this relationship through the ClassCanvas method **/setmenu**. For an explanation of **/setmenu** see "Callbacks, Targets and /setmenu" in this chapter.

Creating Menus

Menus consist of arrays of items. OpenLookMenus can have four types of items: command, submenu, exclusive, and nonexclusive. The visual look, type and callback of menu items are specified by a set of triples specified at the time of menu creation. That is, the triples, one for each item, can be given as arguments to the new method when a menu is instantiated.

The most common way to describe an item is:

[thing|graphic	null|submenu|genproc	proc|null]
visual	type	callback
field	field	field

This way of specifying items can be used to create menus with an arbitrarily large number of items. Item types can be mixed using the above model. For example the following code would specify three types of items in one menu:

```
/mixmenus
        [
                (item1) [/Exclusive] null      % Exclusive item
                (item2) [/Exclusive] null      % Exclusive item
                   (item3) [/Exclusive] null      % Exclusive item
                (item4) mymenu null            % Submenu
                (item5) null {2 3 add pstack}  % command item
        ]
        framebuffer /new OpenLookMenu send
def
```

A less general way of specifying a triple can be used:

[thing|graphic . . .]	null|submenu|genproc	proc|null
visual	type	callback
field	field	field

This way of specifying menus is used to create menus where all the items will be of the same type and have the same callback.

Visual Look Field

You can specify the visual look field of the triple using either a graphic or a thing, both described in ClassGraphic. Whatever graphic is used for the visual field, by default, will be inserted in an instance of OpenLookMenuButton-Graphic.

A thing is a PostScript data structure, either a string or an array. The array contains the string as well as optional attributes used to display the string. Two common attributes used are a font to render the string in and a color to render the font in.

Type Field

The type field is used to indicate the type of the menu. The type field is one of:

- null to specify a command menu item
- a submenu (sublist) or a procedure to create a submenu (genproc)
- an array containing the name /Exclusive to specify an exclusive item
- an array containing the name /Nonexclusive to specify a nonexclusive item

Callback Field

The callback field is null when the item is of type submenu. If the item is not of type submenu then the callback field specifies the action to be taken when the item is selected. If non-null the callback field must be an executable PostScript array that consumes the menu from the stack.

Limitations

The NeWS Development Environment's OPEN LOOK menus do have some limitations and as such, do not currently allow for some OPEN LOOK UI functionality. Menus containing both exclusive and nonexclusive choices are not supported well. For example: a menu is created that has two command items, three exclusive items and one nonexclusive item. One might expect that for the purposes of menu manipulation the menu had four items: the two command items, the non exclusive item and the three exclusive items grouped as a single item. In fact in the NeWS Development Environment, menus cannot handle this grouping of items.

This lack of grouping ability means that a menu cannot contain two sets of exclusive items that operate independently. Turning on any exclusive item in a menu will turn off *all* others.

Laying Out Menus

The default menu layout is one column and the number of rows equal to the number of items. However, you can lay out menus with multiple rows and columns. Moreover, you can specify whether the menu will be layed out in row major or column major order.

To set the layout of a menu instance to be other then the default use **/setlayoutstyle. /setlayoutstyle** takes a RowMajor? boolean first argument; true lays out the menu as row major and false lays out the menu as column major. The next two arguments specify the number of rows and the number of columns respectively; either or both may be null.

If you specify the number of rows and/or columns, then those number of rows and/or columns are created. If you specify null for one of the arguments then the appropriate value is calculated automatically by dividing the number of items to be displayed by the other, known value. If null is specified for both rows and columns the menu is layed out in the default style.

If you specify a matrix that is too large for the number of items there will be empty spots in the matrix. If too many items are specified for the size of the matrix the excess items will not be shown.

You can query the layout specifications of a menu using **/layoutstyle**. The RowMajor? boolean, the number of rows, and the number of columns are returned.

Manipulating Menus

Once you have created a menu several methods are available to change features or query current menu settings. Some of the more commonly used methods allow a program to insert, delete, disable and determine the last item selected on the menu.

You can access menu items using their index. The terms location and index are used interchangeably.

Use **/insert** to put a new item into an existing menu at a specified location. **/insert** takes an index and an item triple as arguments and preserves the menu's default. Similarly, **/delete** removes an item at the specified location while preserving the menu's default selection. Any field of an item's triple can be changed using **/change**. If all the fields aren't being changed then null is used as placeholder when the triple is passed into **/change**.

Enabling and disabling an item refers to two things: the visual state of the item and the ability of the item to get the combination of mouse drags and button ups. An item that is enabled will be highlighted on a mouse drag and execute its callback on the button up. An item that is disabled will get neither the mouse drag or the button up. A disabled item is "grayed out" to distinguish it from an item that is enabled.

Use **/enableitem** to enable a menu item and **/disableitem** to disable an item. The **/itemenabled?** method allows you to query an item to determine if it is enabled or disabled.

Menu Values

The value of a menu is the index of the last item selected. Send **/value** to a menu instance to determine its current value.

Use **/setvalue** to change the value of a menu. If an item is an exclusive item *and* it is turned on when **/setvalue** is called on it, the item is not turned off. Use **/nonxvalue** to determine which nonexclusive menu items are turned on. It returns an array of the indices of all the nonexclusive items currently set. If no nonexclusive values are set an empty array is returned. Use **/xvalue** to determine which exclusive value is currently set. It returns the index of the item. **/setvalue** does not execute an item's callback. Applications that want an item's callback executed immediately after the item is set should call **/doaction**.

Pinned Menus

It is worthwhile noting that there are no programmer interfaces to allow access to the pinned version of the menu. The pinned version is a copy of the menu. Code in class OpenLookMenu will perform the necessary actions to keep the pinned copy up-to-date with changes made to the menu.

To give menus a pin (or to remove it) use **/setpinnable**. Use **/pinnable?** to query a menu to determine if it is pinnable.

Callbacks, Targets and /setmenu

After a menu is built several conditions must be met before the "right" thing can happen when a menu item is selected. First a menu and a canvas must be associated so that MENU down produces the correct menu. Second, when a MENU up occurs the correct callback is executed. Finally when the callback is executed the correct object is affected. The NeWS Development Environment provides several methods and a mixin class to help you manage the relationship between menus and other canvases.

You can use the ClassCanvas method **/setmenu** to associate a menu with another canvas. **/setmenu** takes a menu as its argument and is sent to a canvas.

Using **/setmenu** causes the canvas to express an interest in the MENU button down and to display the menu when one is noticed. When **/setmenu** is used menu callbacks will execute in a process forked from the canvas's event manager process. Having the callback execute in the canvas's event manager process group guarantees that the same userdict and stdout (for sending tagprints), exists when the callback is executed as when the canvas was activated.

However, before the correct callback can be executed a program must determine which item was selected. Determining which item was selected is easy because when an item is selected the menu is pushed on the stack.

With the menu on the stack programs can call **/value** to determine which menu item was selected. **/value** returns the index of the last item selected. Thus in the simplest case to determine which item was selected a callback would contain:

```
/value 1 index send
```

With the item selected identified the correct callback can be executed. Then the callback needs to send its methods to the correct object.

The NeWS Development Environment provides targets to help programs send callback methods to the correct object. If **/setmenu** was used to associate the menu with a canvas then the ClassTarget method, **/sendtarget** can be used to send the callback methods to the canvas. **/sendtarget** is sent to the menu. Thus in the simplest case the callback would contain:

```
/mymethod /sendtarget 3 -1 roll send
```

/setmenu associates a menu with another canvas; it doesn't set the target. Instead, by default canvases set the target of their menus to be themselves when the menu is brought up over them (see Chapter 3, Canvases).

Using Targets Manually

While most applications will want the Toolkit to manage menus, functionality is provided to allow applications to handle menus manually. If you require manual handling of menu display, e.g., having different menus pop-up in different areas of the same canvas, should not use **/setmenu**.

Such applications could, for example, express interest in MENU mouse down, calculate where in the canvas the mouse was when the mouse down was received, then make an explicit call to **/showat** to display the correct menu for that region of the canvas.

Targets can be set explicitly using **/settarget**. In addition, applications that don't want the canvas over which a menu is brought up to be the target of the menu can change the default behavior by using the ClassCanvas method **/autotarget-menu:**

```
false /autotargetmenu mycanvas send.
```

See Chapter 8 for a complete discussion on using targets.

Shared Menus

Since canvases are the default target of the menus brought up over them, applications can easily share the same menu between different instances of a canvas. There are two cases:

1. All instances of a class share the same menu. Use /CanvasMenu a ClassCanvas class variable:

```
/CanvasMenu
        <menu-definition>
def
```

2. Share a common menu between instances of different classes. Then an application would use **/setmenu** and send the menu to each canvas instance.

Settings

OpenLookXSetting

This class implements the Open Look exclusive setting control. Since this class is derived from ClassSelectionList, entire group of settings uses a single canvas, has a single event manager.

Although not derived from ClassControl, exclusive settings behave much like controls. They have a single value and a client-supplied notify proc. They may be placed into bags along with controls. However, a difference is that although individual items may be disabled, there is no /disable method that applies to the entire setting group.

Exclusive settings are created from a list of items. The list is similar to the one used to create menus, except there is nothing corresponding to a submenu. There are two forms of parameter to **/new**:

```
[thing|graphic proc|null ...] canvas
[thing|graphic ...] proc|null canvas
```

The first form is an array of graphic/proc pairs. Each graphic may be either an instance of a graphic class, or may be a "thing" suitable to create an instance of class OpenLookXSettingGraphic. The proc is a notify proc like that of any menu or control. It is called with the exclusive setting on the operand stack. The second form uses a single proc for all items and is used when the same notify proc is used for all setting items.

As with menus, the items in a setting are ordered. The value of the setting control is an integer corresponding to the selected item. The first item is numbered 0. The ClassSelectionList methods for managing lists can be used to insert, delete and change items. Individual items can be enabled and disabled.

Methods: (all in ClassSelectionList)
/insert
/delete
/change
/enableitem
/disableitem
/itemenabled

OpenLookXSetting uses a row/column layout algorithm to arrange its choices (it has class RowColumnLayout as a superclass as well as ClassSelectionList). The /setlayoutstyle method is used to control whether the exclusive choices are arranged in a row, a column or a matrix.

Methods: (all in class RowColumnLayout)
/setlayoutstyle **/layoutstyle** **/cellcount**

Example

This code builds a demo frame that contains a bag that contains a simple exclusive setting. When an item is selected, the frame's footer displays the new setting.

```
/frame /demo OpenLookBaseFrame send def
% Create the bag and put it in the frame.
/bag framebuffer /new RowColumnBag send def
bag /setclient frame send pop
% Create the exclusive setting.
[     (300)  (1200)  (2400)  (4800)
      (9600) (19200) (57600)]               % Setting choices
{
      /valuething 1 index send null         % setting string null
      /setfooter /sendtarget 5 -1 roll send % -
}                                           % Setting notify proc
framebuffer                                 % Parent canvas
/new OpenLookXSetting send
/xsetting exch def

% Set the setting's target to be the frame.
frame /settarget xsetting send

% Shape the setting to its minimum size.
0 0 /minsize xsetting send /reshape xsetting send

% Put the setting inside the bag.
/XSetting xsetting /addclient bag send

% Make sure the setting is displayed.
/paint bag send
```

Notes:

The creation of the frame and bag are kept very simple for this example. A RowColumnBag is used so we don't have to provide any information about the position of the setting.

It is not necessary to /activate the setting when it is put inside a bag -- the bag does this automatically.

The **/settarget** method is used to make the setting's target be the frame. The setting's notify proc uses **/sendtarget** to call the **/setfooter** method in the frame. The **/sendtarget** method is sent to the setting, which is on the operand stack when the notify proc is called.

The **/valuething** method returns the "thing" corresponding to the current value of the setting control. This will be one of the strings (300), (1200), etc.

The bag is explicitly painted at the very end of the example because the frame and bag were already activated earlier in the example. (The frame is activated by the **/demo** method.) In many actual cases the bag will be populated with controls before its frame is activated. When the frame is activated the bag and its controls will be painted automatically, eliminating the need for an explicit call to /paint.

The default layout style is to place the setting items in a single column. This may be changed with the /setlayoutstyle method. For example, the following will change the above setting control from vertical to horizontal:

```
% Make layout be by rows, with one row and as many columns
% as needed.
true 1 null /setlayoutstyle xsetting send

% Reshape the setting control.
0 0 /minsize xsetting send /reshape xsetting send

% Invalidate and repaint the bag.  Invalidation causes the bag
% to relayout its contents.  This is necessary because of the new
% shape of the setting control.
/invalidate bag send
/paint bag send
```

OpenLookChoggle

An OpenLookChoggle is a variation on the OpenLookXSetting in which it is possible for no item to be selected. If the user clicks the mouse on the selected item, it is deselected and no other item is selected. The /value method will return null for the control when no item is selected. Other than this difference, OpenLookChoggles are exactly the same as OpenLookXSettings. The name "choggle" is a blend of "choice" and "toggle" and was once used in the Open Look specification to as the name for what is now called simply a "variation on exclusive settings".

OpenLookNonXSetting

A nonexclusive setting is a group of choice items, any number of which may be selected at the same time. Clicking on an item toggles its selected state. This is a multiple valued control -- its value is an array of values corresponding to the selected items. The /setvalue method takes either a single value of an array of values, /value returns an array of values, and /valuething returns an array of things.

Methods:
/value **/setvalue** **/valuething**

The spacing between the setting choices is 4 by default. This may be changed with the /setgaps method.

Class OpenLookNonXSetting is a subclass of OpenLookXSetting. Except for their having mulitple values and the spacing of their choices, nonexclusive settings are identical to exclusive settings. Their settings are specified by the same parameter to /new, items may be inserted, deleted or changed the same way, and the layout style is controlled the same way.

Example

This example creates a nonexclusive setting and adds it to the bag from the previous example.

```
% Create the nonexclusive setting.
[(Bold) (Italic) (Underline) (Overstrike)]      % Setting choices
{
    /valuething 1 index send                    % setting stringarray
    () exch {append ( ) append} forall          % setting string
    null exch                                   % setting null string
    /setfooter /sendtarget 5 -1 roll send       % -
}                                               % Setting notify proc
framebuffer                                     % Parent canvas
/new OpenLookNonXSetting send
/nxsetting exch def

% Set the setting's target to be the frame.
frame /settarget nxsetting send

% Shape the setting to its minimum size.
0 0 /minsize nxsetting send /reshape nxsetting send

% Lay out the bag in rows, so the nonexclusive setting will
% be to the right of the exclusive setting, not below it.
true null 2 /setlayoutstyle bag send

% Add the setting to the bag.
/NonXSetting nxsetting /addclient bag send

% Make sure the setting is displayed.
/paint bag send
```

Notes:

This example assumes the exclusive setting from the previous example is still laid out as a column. If it is a row, the example still works, but doesn't look as good.

The /valuething method returns an array of things for the nonexclusive setting. Each thing is a string such as (Bold) or (Italic). A single string is built by appending together all the strings in the array. The resulting string is used to change the right part of the frame's footer.

The layout style for the bag is set in this example because a second client is being added to the bag and therefore it matters whether the bag layout is in a row or a column.

Here are some examples of setting and retrieving the value of the nonexclusive setting.

```
% Select no items.
null /setvalue nxsetting send
/value nxsetting send                          % []

% Select one item.
1 /setvalue nxsetting send
/value nxsetting send                          % [1]
/valuething nxsetting send                     % [(Italic)]

% Select several items.
[0 2] /setvalue nxsetting send
/value nxsetting send                          % [0 2]
/valuething nxsetting send                     % [(Bold) (Underline)]
```

6. CONTROLS

6. CONTROLS

6 Controls

Controls

Controls 6-1
Introduction 6-1
ClassControl 6-1
- Value 6-1
- Notification 6-2
- Enabled / Disabled State 6-3
- Tracking 6-3

ClassDialControl 6-5
- Deltas 6-5
- Normalization 6-6

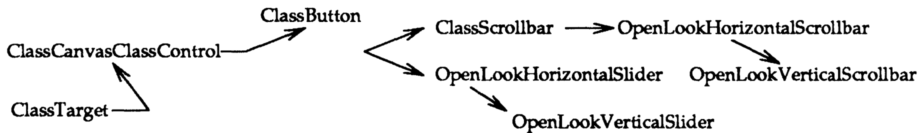
ClassButton 6-7
- Graphic 6-7
- Notification and Value 6-8
- OpenLookButton 6-8
- Button Examples 6-8
- OpenLookButtonStack 6-11
- OpenLookAbbrButton 6-14
- OpenLookAbbrButtonStack 6-14

Analog Controls 6-16
- Sliders 6-16
- Scrollbars 6-17
- Simple Scrollbar Example 6-19

Fields 6-20
- ClassTextControl 6-20
- OpenLookTextControl 6-22
- OpenLookNumeric 6-22

Controls

Introduction



ClassControl

Controls are canvases with the following features:

- a value
- a notify (callback) procedure
- a state of enabled or disabled
- a tracking process that is created for user interaction

Value

A control has a "value" that may be changed via a user interaction or programmatically. The general mechanism provided by ClassControl allows a control's value to be any object. Subclasses will generally define some domain of legal values for the controls they define. For example, a check box might allow only the values true and false, a dial might allow integers from 0 to 10, and a text field might allow any character string. Each subclass defines what happens if an attempt is made to set the value to something not in the domain of legal values. A subclass might allow multiple values, in which case the control value might actually be an array of the current values.

A subclass of ClassControl must provide a **/PaintValue** method. This method is called whenever the control's value changes. The method may either paint the control according to the new value, or may compare the new value to the old and do an incremental paint.

Methods:
/value
/setvalue
/PaintValue

Notification

If a control's value changes as the result of an interaction initiated by the user, the client application is informed via the "notify" procedure. This is a fragment of code that is supplied as an argument to **/new** for the control, or may be set with the **/setnotifyproc** method. The notify proc is called with the control itself on the operand stack. The notify proc can query the control for its value or any other relevant information. The notify proc must remove the control from the stack. It is generally best to not rely on a particular execution context from within the notify proc. Here's a sample notify proc that prints the current value of the control:

```
/mynotify { % control => -
        /value exch send
        console (Control value is %0 [4 -1 roll] fprintf
} def
```

The methods **/callnotify** and **/checknotify** cause notification to take place, i.e. the control is pushed onto the operand stack and the control's notify procedure is executed. The **/callnotify** method unconditionally notifies. The **/checknotify** method calls /CallNotify? and uses the returned boolean value to decide if **/callnotify** should be called. Generally, you should call **/checknotify**, not **/callnotify**. The /CallNotify? method can be overridden by a subclasser to control when notification takes place. The default **/CallNotify?** method checks to see if the current value of the control is different from the value at the last notification, and if so the notification takes place.

Some controls are output-only and therefore do not support user interactions and do not do automatic notification. The only way notification can take place for these controls is via an explicit call to **/checknotify** or **/callnotify**. An example is a read-only text control.

All controls take a notify proc as an argument to their **/new** method. If the argument is null, no notification takes place. Since controls are canvases, they also take a parent canvas as an argument to new. Control subclasses will often define additional parameters to **/new**.

Methods:
/new
/setnotifyproc
/notifyproc
/checknotify
/callnotify
/notifiedvalue
/CallNotify?

Enabled / Disabled State

A control has an enabled/disabled state that is changed by the **/enable** and **/disable** methods. A disabled control is "read only" and does not respond to user input. Typically a disabled control will offer a visual indication of its state, such as dimming itself.

Methods:
/enable
/disable
/enabled?
/PaintEnabledState

Tracking

"Tracking" refers to the temporary processing of certain events during user interaction with a control. It is typically initiated by the user clicking the mouse in the control. At that time certain tracking interests are expressed, allowing processing of, for example, mouse button up events to terminate tracking and mouse enter/exit events for highlighting the control.

Controls automatically express an interest in the down transition of PointButton (defined by *The OPEN LOOK UI* as the left mouse button, if not overridden by the user.) This causes the **/EventHandler** method to be called, which by default does nothing. A subclass will typically override this method to call **/StartTracking**, which will call **/trackon** if the control is enabled. The **/trackon** method

creates a track manager process (accessable via the **/trackmgr** method) and expresses the transient tracking interests.

There are several "standard" tracking methods that, if supplied by a control subclass, will be called automatically during tracking. If **/ClientDrag** is supplied, it will be called for every mouse movement within the control. If **/ClientEnter** or **/ClientExit** is defined, it will be called when the mouse cursor enters or leaves the control. The **/ClientDown** and **/ClientUp** methods are called at the start and end of tracking.

If a **/ClientRepeat** method is provided, a timer interest is expressed and the method will be called after a time interval specified by ClientStartTime. The **/ClientRepeat** method will typically take some action, such as calling **/ClientDown**, and then generate another timeout event so it will be called again:

```
/ClientRepeat { % event => -
        ClientRepeatTime /canvas self send
        currentprocess sendtimeoutevent        % -
} def
```

Note that the event passed as argument to **/ClientRepeat** will not have meaningful /XLocation and /YLocation fields. The event's /Name will be /TimeOutEvent rather than the name of the event that started tracking.

Methods	
/trackon	**/ClientDrag**
/trackoff	**/ClientEnter**
/trackmgr	**/ClientExit**
/trackinterests	**/ClientRepeat**
/StartTracking	**/ClientStartTime**
/EndTracking	**/ClientRepeatTime**
/ClientDown	**/EventHandler**
/ClientUp	**/MakeTrackInterests**
/BuildTrackInterest	

ClassDialControl

A dial is a control with a numerical value bounded by minimum and maximum values. Examples of dials are circular knobs and meters, linear scrollbars and thermometer-type gauges.

User interaction with the dial may change its value either to an absolute value, or to a value computed by adding or subtracting a "delta" to the current value. Values are constrained to be within the dial's "range", i.e. between its minimum and maximum values. The granularity of the dial may be controlled via "normalization". For example, a dial may have a range of 0 to 100 and normalization may be used to contrain legal values to multiples of 10. Normalization values will generally be less than or equal to delta values.

Methods:
/setrange
/range
/CheckValueBounds

Deltas

Deltas are named increments for amounts a dial's value may change relative to its current value. Deltas are defined in subclasses of ClassDialControl. For example, a scrollbar may have deltas named /Line and /Page. A delta may be a constant value or an executable code fragment that produces a dynamically computed value.

The **/setdelta** method allows you to define new delta names and values for a control. The **/incrementvalue** method takes an integer and a delta name and changes the dial's value by the delta amount multiplied by the integer you supply. The **/motion** method returns the amount and delta name of the last change in the dial's value. It is most useful when called from the dial's notify proc.

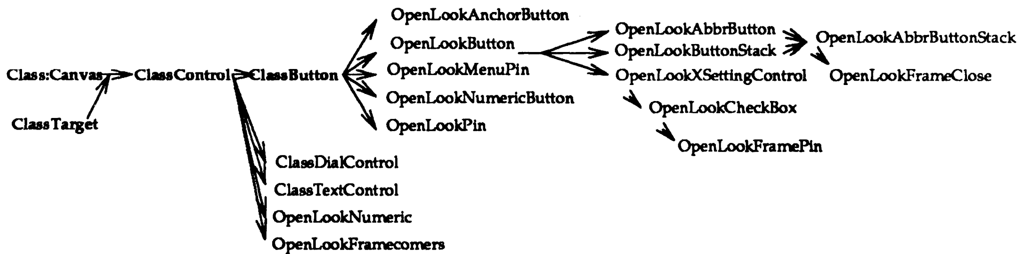
Methods:
/setdelta
/delta
/motion
/incrementvalue
/SetMotion

Normalization

The normalization value, which may be either constant or dynamically computed, specifies the difference between consecutive legal values of the dial. For example, a dial with a minimum value of 0, a maximum value of 100 and a normalization of 2 could have only even values from 0 to 100. Delta values will usually be some integral mulitple of the normalization value. Normalization is most commonly used to contrain the values a dial may acquire from an absolute motion, rather than from a relative motion involving a delta.

Methods:
/setnormalization
/normalization
/Normalize

ClassButton



A button is a very simple control, having a graphic and a boolean value. When the value is true, the button is highlighted. When the mouse is clicked on the button, tracking is started. When the mouse cursor is inside the button, the button is highlighted. When the mouse button is released, the button's notify proc is called, and the button's value is set back to false, causing it to be unhighlighted.

Graphic

A button contains a graphic, which determines the button's appearance. When you create a button, you must supply either an existing graphic instance, or a "thing", from which a graphic may be created. If you supply a terminal graphic, it determines the button's appearance completely. If you supply a thing or a non-terminal graphic, it is enclosed inside another graphic that supplies the button border.

Methods:
/new /setgraphic /graphic /EnGraphic /UnGraphic /CreateGraphic

Notification and Value

When you use a button you are generally interested in the notification that takes place when the button is pressed but not the button's value. The notify proc gets called when the mouse button is released on the button. The value of the button is true when it is highlighted and false otherwise; so the value will always be true during the notification.

OpenLookButton

Class OpenLookButton implements a button with OPEN LOOK appearance. As with all buttons, the appearance of the button is specified via a "thing" or graphic parameter to /new.

Button Examples

The simplest possible button has a trivial notify proc and has the framebuffer as its parent canvas:

```
(Do It)                          % Button label
{pop console (Done!) fprintf}    % Notify proc
framebuffer                      % Parent canvas
/new OpenLookButton send
/button exch def

% Shape the button to its minimum size.
100 100 /minsize button send /reshape button send

% Start event processing - make button mouse sensitive.
/activate button send

% Actually paint the button.
/paint button send
```

Notes:

The notify proc is called with the button itself on the operand stack. Since the "Done!" message does not use the button at all, the button is simply popped from the stack.

The **/paint** method call is necessary because the button has the framebuffer as its parent canvas. Since the button is a transparent canvas and the framebuffer does not automatically ask transparent children to paint themselves, the /paint must be explicit. If the button had been placed inside a bag, the **/paint** would not be needed because bags automatically paint their child canvases.

A slightly more involved and realistic example puts the button inside a bag which is inside a frame.

```
% Create the frame.
/frame /demo OpenLookBaseFrame send def

% Create the bag and put it in the frame.
/bag framebuffer /new RowColumnBag send def
bag /setclient frame send pop
(Repaint Frame)                       % Button label
{/paint /sendtarget 3 -1 roll send}   % Notify proc
framebuffer                           % Parent canvas
/new OpenLookButton send
/button exch def

% Set the button's target to be the frame.
frame /settarget button send

% Shape the button to its minimum size.
0 0 /minsize button send /reshape button send

% Put the button inside the bag.
/RepaintButton button /addclient bag send

% Make sure the button is displayed.
/paint bag send
```

Notes:

The creation of the frame and bag are kept very simple for this example. A RowColumnBag is used so we don't have to provide any information about the position of the button.

It is not necessary to /activate the button when it is put inside a bag—the bag does this automatically.

The **/settarget** method is used to make the button's target be the frame. The button's notify proc uses **/sendtarget** to call the **/paint** method in the frame. The /sendtarget method is sent to the button, which is on the operand stack when the notify proc is called. Another technique for repainting the frame would be to send **/paint** to the parent of the parent of the button (the button's immediate parent is the bag). This is somewhat messier and less flexible than using a target.

OpenLookButtonStack

An OpenLookButtonStack is an OpenLookButton that has a menu associated with it. A different graphic (class OpenLookButtonStackGraphic) is used so the button contains an arrow to indicate there is a menu. The name "stack" is an artifact of an earlier Open Look revision in which what is now called a "menu button" was called a "button stack".

The **/new** method includes a specification for the menu: either a menu object itself or an array that can be used to create the menu. If the parameter is an array, it is used to instantiate the menu, and has the following form:

```
[thing|graphic genproc|sublist|null proc|null ...]
```

Menus created in this way have the framebuffer as their parent. The **/setmenu** method allows you to change the menu associated with a button stack, and, like **/new**, it takes either a menu instance or a menu specification array as its argument.

Methods:
/new **/setmenu**

Notification

Notification for OPEN LOOK button stacks is different than for most other controls. For most controls if you do not provide a notify proc, no notification will take place. However, for button stacks, if you provide no notify proc, pressing the button will cause the notify proc for the default item on the associated menu to be executed. For most cases this is the desired behavior so the notify proc parameter to /new for the button stack will be null.

Methods:
/NotifyUser

OPEN LOOK button stacks display the menu default in the button when the button is pressed. This is implemented by the **/DisplayDefault** and **/UnDisplayDefault** methods, which may be overridden by subclassers.

Methods:
/DisplayDefault
/UnDisplayDefault

Button Stack Example

```
% Create the frame.
/frame /demo OpenLookBaseFrame send def

% Create the bag and put it in the frame.
/bag framebuffer /new RowColumnBag send def
bag /setclient frame send pop

(Frame)     % button label
[
            (Close)     null {false /open       /sendtarget 4 -1 roll send}
            (Zoom)      null {true  /zoom       /sendtarget 4 -1 roll send}
            (Unzoom)    null {false /zoom       /sendtarget 4 -1 roll send}
            (Repaint)   null {/paint            /sendtarget 3 -1 roll send}
            (Flash)     null {/flashframe       /sendtarget 3 -1 roll send}
            (Front)     null {/totop            /sendtarget 3 -1 roll send}
            (Back)      null {/tobottom         /sendtarget 3 -1 roll send}
            (Reshape)   null {/reshapefromuser  /sendtarget 3 -1 roll send}
]
null                                             % Button notify proc
framebuffer                                      % Parent canvas
/new OpenLookButtonStack send
/buttonstack exch def

% Set the targets for the button stack and the menu to be the frame.
frame /settarget buttonstack send

% Shape the button to its minimum size.
0 0 /minsize buttonstack send /reshape buttonstack send

% Put the button inside the bag.
/ButtonStack buttonstack /addclient bag send

% Make sure the button is displayed.
/paint bag send
```

Notes:

The button stack menu performs various operations on a frame, which is established as the menu's target. The menu has no submenus, so that parameter is null for each menu item. Some of the frame methods take arguments, which are included in the menu item notify procs.

The notify proc for the button itself is omitted, i.e. is null. When the user clicks on the button, the notify proc for the default menu item is called. The notify proc for the button itself is omitted, i.e. is null. When the user clicks on the button, the notify proc for the default menu item is called.

For button stacks, the target for the menu is automatically set to the target for the button stack. It is not necessary to explicitly set the target for the menu; only the button stack target is set in the example.

OpenLookAbbrButton

Class OpenLookAbbrButton implements a small square button with an arrow inside it and an optional label to its right.

The button label may be an arbitrary graphic. The argument may be a "thing", in which case it is used to create an instance of class OpenLookLabelGraphic. If no graphic is desired, the parameter to **/new** should be null.

The direction that the button's arrow points may be changed via the **/setarrow** method, which takes arguments /Left, /Right, /Up and /Down. The default arrow direction is down.

OpenLookAbbrButtonStack

Class OpenLookAbbrButtonStack combines OpenLookAbbrButton and OpenLookButtonStack to produce an an abbreviated button, a label, and a menu. The button's label reflects the most recent selection from the menu. When a menu item is chosen, the item's notify proc is called and the button stack label is changed to match the menu item. As with button stacks, the menu default is previewed in the button's label when the button is pressed. When the button is released, the label reverts to its previous value, and the button stack notify proc is called. If there is no notify proc for the button stack (the parameter to **/new** was null), the notify proc for the default menu item is called.

The button stack's label, menu, notify proc and parent canvas are specified as paramters to **/new**. The label is specified as a "thing" or as a graphic. The parameter may be null, in which case no label is displayed, either initially or when a menu item is selected. The menu is specified the same as for class OpenLookButtonStack. The notify proc is the proc called when the abbreviated button is pressed, and may be null. Remember that each item in the associated menu may also have its own notify proc.

Example

```
/frame /demo OpenLookBaseFrame send def

% Create the bag and put it in the frame.
/bag framebuffer /new RowColumnBag send def
bag /setclient frame send pop

% Create the menu to use with the button stack.
[(Icons) (Colors) (Menus) (Text) (Messages)
(Mouse Settings) (Keyboard Equivalents)]                    % Menu items
[/Exclusive]                                                % Exclusive choice menu
{                                                           % Notify proc
        console (% selected.0
        [/valuething 5 -1 roll send] fprintf
}
framebuffer                                                 % Menu parent
/new OpenLookMenu send
/menu exch def

% Set menu default to "Text"
3 /setdefault menu send

% Create the abbreviated button stack.
(Text                               )                       % Initial label
menu                                                        % Stack menu
null                                                        % Button notify proc
framebuffer
/new OpenLookAbbrButtonStack send
/buttonstack exch def

% Shape the button to its minimum size.
0 0 /minsize buttonstack send /reshape buttonstack send

% Put the button inside the bag.
/ButtonStack buttonstack /addclient bag send

% Make sure the button is displayed.
/paint bag send
```

Notes:

The menu is created and then used as a parameter when the button stack is created. This is necessary in this example so the menu may contain exclusive choices. If "normal" menu items were wanted, it would be possible to create the menu implicitly with an array parameter to the **/new** method of OpenLookAbbrButtonStack.

The initial button label, "Text", is padded with blanks so the label graphic that is created will be large enough to accommodate the largest menu item label, "Keyboard Equivalents". This is really just a workaround for a toolkit bug.

The menu callback simply prints the selected menu item label on the console. In an actual application the callback would probably use **/sendtarget** to call a method in some object associated with the menu.

The following variation on the menu notify proc causes the menu's default to be changed to the menu item following the current choice every time a menu item is chosen. Since clicking on the abbreviated button chooses the default menu item, the effect of this code is to build an abbreviated menu button that steps through the menu choices sequentially.

```
{       % Menu notify proc
        console (% selected.0
        [/valuething 4 index send] fprintf
        /value 1 index send 1 add               % Menu value + 1
        dup /maxlocation 3 index send gt {      % Check for cycle
                pop 0
        } if
        /setdefault 3 -1 roll send
}
```

Analog Controls

Sliders

Sliders are a simple subclass of ClassDialControl. They implement a subset of OPEN LOOK sliders as described in the *OPEN LOOK UI Specification.* A slider only defines one delta, /Line. There's currently no way to show the users what the actual value of the slider is, because there is no way to display the scale. There are no tick marks, and there is no way to tell what the minimum and maximum values of the slider are. You can get some of this functionality if you are willing to write some postscript code. You will have to write some code to display the current, minimum and maximum values of the slider with numeric controls. But you will have to use a FlexBag or your own subclass of ClassBag to group them together in a reasonable way.

The following code will create and activate a slider:

```
{ % slider => -
        (Slider value = %0 [ /value 4 -1 roll send ] printf
} framebuffer /new OpenLookHorizontalSlider send            % slider

1 20 /setrange 3 index send                                 % slider
[/Line 1] /setdelta 2 index send                            % slider
1 /setnormalization 2 index send                            % slider
4 /setvalue 2 index send                                    % slider

10 30 610 /minsize 4 index send exch pop                    % slider x y w h
/reshape 5 index send                                       % slider
/activate 1 index send                                                % slider
/map 1 index send                                           % slider
/paint 1 index  send                                        % slider
```

The slider's callback simply prints the current value of the slider. It could be modified to display the new value in a numeric control.

Scrollbars

ClassScrollbar is a descendant of ClassControl, and like all controls, has a target at which it directs certain actions. A typical target, or client, for a scrollbar is some sort of text canvas, a canvas that knows how to display text and scroll through it. A scrollbar must be given a target to scroll, a callback, and must have certain parameters (called deltas) set to values appropriate for that application (see below for an example). The scrollbar will handle painting itself, updating its position and value, and will automatically call its callback when a change has occurred. It is the responsibility of the callback to inform the scrollbar's target of the type of action that has occurred, e.g., scroll one line, or one page, or to the end of the document.

ClassScrollbar is a subclass of ClassDialControl. It defines some new deltas appropriate to scrollbars. The scrollbar deltas are named /Line, /Page and /Document. They specify how much to increase or decrease the value of the scrollbar when a user selects a particular type of scrolling. For example, values for these deltas might be specified in units of lines, and so /Line would be set to 1, /Page would be the number of lines visible in a page, and /Document would be the number of lines in the document. These deltas are set by the scrollbar's target.

Scrollbars know how to scroll by lines, by pages, to absolute positions, and to the beginning and end of whatever's being scrolled. These types of scrolling correspond to motions defined by scrollbars. The motion names are /Line, /Page, /Document and /Absolute.

When the scrollbar's callback is called, it should query the scrollbar for the type of motion that occurred with the **/motion** method. **/motion** returns a value and the motion name. The callback uses the value and the motion name to update the target appropriately. How the value is interpreted depends on the kind of motion. When the motion is /Absolute the value is the current value of the scrollbar and the target should arrange to scroll to that position. Otherwise, the value is either 1 or -1, depending on whether the scrollbar moved forward or backward, respectively, a line or page, or to the beginning or end of the document.

There is another special purpose delta called /View. The scrollbar uses /View in conjunction with the /Document delta to display the proportion indicator. /View defaults to the current value of the /Page delta.

While a scrollbar knows about its target, that target will often also want to know about its scrollbar. For instance, a text editor will want to update the scrollbar's deltas whenever some lines have been inserted or deleted, or set a new value when some user action in the editor causes the document to be scrolled to a different place, e.g., searching.

ClassScrollbar is an abstract class; that is, it is never instantiated itself, but rather is subclassed. To create a scrollbar, use OpenLookHorizontalScrollbar or OpenLookVerticalScrollbar. They are subclasses of ClassScrollbar, and all they do in the subclass is handle the OpenLook scrollbar look and feel.

Simple Scrollbar Example

```
% Create a scrollbar with a callback.
%
/scrollbar { % sbar => -
        /motion 1 index send                        % sbar value motion-name
        {                                           % sbar value
        /Absolute { /scrollabsolutely }
        /Line { /scrollbylines }
        /Page { /scrollbypages }
        /Document {
                dup -1 eq {
                        /scrolltobeginning
                } {
                        /scrolltoend
                } ifelse
                }
        } case                                      % sbar value methodname
        /sendtarget 4 -1 roll send
} framebuffer /new OpenLookVerticalScrollbar send def

/ScrolledObject Object []
classbegin
        /scrollabsolutely   { pop (Absolutely) = } def
        /scrollbylines      { pop (By lines) = } def
        /scrollbypages      { pop (By pages) = } def
        /scrolltobeginning  { pop (To beginning) = } def
        /scrolltoend        { pop (To end) = } def
classend def

/foo /new ScrolledObject send def
foo /settarget scrollbar send
100 100 20 300 /reshape scrollbar send
/activate scrollbar send
/map scrollbar send
/paint scrollbar send
```

This sample code uses a simple class called ScrolledObject, which understands methods for scrolling. This particular object defines methods which just print out a message saying that it was called, but in a real life application they would actually do something more useful. This example creates a scrollbar, then creates one of these scrolled objects, and makes it the target of the scrollbar. Then it just reshapes, activates, maps and paints the scrollbar. The callback for the scrollbar simply queries the scrollbar (which is passed as an argument to the callback) for the type of motion that occurred, and then performs a case

statement to figure out what method to call, and then it sendtarget's that method to the scrollbar's target.

Fields

ClassTextControl

A text control is a control whose value is a string. Text controls several methods of ClassControl; e.g., **/CallNotify?** and related methods do a string comparison rather than a simple 'eq' to determine if the value has changed since the last notification.

A text control also registers its canvas as an input focus client; if it receives the focus, keystrokes are treated as characters to be inserted into the control's value. Certain keystrokes, such as backspace, cause other modifications to the control's value. The text control calls **/checknotify** (and thus **/callnotify** if the value has changed) whenever it loses the input focus, or when the RETURN key is pressed; it does not notify on each keystroke.

Text controls are also selection clients. Thus they do not do tracking in the same way as other controls; they do not override the **/EventHandler** method, nor should subclassers do so. Instead they express a Selectable interest, and the global selection manager handles the appropriate tracking. Much of the code in ClassTextControl is for handling selections; writers of other text-selection code may wish to look at ClassTextControl for guidance.

Read-only Text

In addition to the enable/disable methods, it is possible to define a text control as being read-only. This has much the same effect as disabling the control, the main difference being how it paints. A disabled control may paint dimmed, whereas the read-only control paints normally but simply refuses to accept user input.

Methods:
/setreadonly

Operating on the Text

Text controls offer several methods for programmatically altering the text string. The inherited method **/setvalue** replaced the entire string. The other methods operate at the current insertion point, which can be set by the user via the mouse, and can also be set programmatically by **/setposition**. (There is unfortunately no position method; clients can obtain the current position by sending /Left, which yields the number of characters left of the caret.)

In addition to modifying the text, clients can also change which character appears at the left edge of the control; characters further to the left are clipped. This can be used to scroll the text to keep the caret within the visible region. Many of the text-modifying methods conclude by sending **/FitCaret**, a subclasser method that can be overridden to specify that the caret should remain visible. The default **/FitCaret** never scrolls the text, but this is overridden by OpenLook-TextControl (see below). The client method **/fitcaret** calls **/FitCaret** with no other changes.

Methods:	
/setposition	**/delword**
/fitcaret	**/scroll**
/inserttext	**/Left**
/delchar	**/FitCaret**
/delspan	

Appearance

The number of characters that can be displayed in a text control depends on the size it is given. The **/minsize** method for text controls requests a size based on the number of characters the control is expected to contain. The default is 5 characters, but this can be changed for any given control (or for a subclass).

Much of the visual behavior of a text control is inherited from ClassCanvas. Two methods of particular interest are listed here.

Methods:
/setdisplaychars
/settextparams
/setcolors

OpenLookTextControl

The OPEN LOOK subclass of ClassTextControl adds no new client methods, but simply overrides some existing methods to provide OPEN LOOK functionality. Specifically, it paints scroll buttons when the text extends beyond the end of the control, it paints a line just under the baseline of the text, and it overrides **/FitCaret** to keep the caret visible after most operations. OpenLookTextControl is the default class for ClassTextControl; i.e., it is obtained by sending **/newdefault** to ClassTextControl.

OpenLookNumeric

There is no "intrinsic" numeric control; there is only the OPEN LOOK form. A numeric control combines a text control with a pair of buttons that modify the numeric value by a specified increment, which is initially 1 but can be changed with **/setincrement**. (If the increment is set to zero, the increment/decrement buttons are removed.) The **/value** of a numeric control is a PostScript number (integer or real). It is by default restricted to the range -32768 to +32767, but this range can be changed by calling **/setrange** (or the individual methods **/setmin** and **/setmax**).

Non-numeric characters can be typed into the text control, but will result in the value being replaced with zero the next time the notifyproc is called. (This occurs the same as for any other text control: on the RETURN key or loss of input focus, if the contents have been changed.)

Methods:
/setincrement
/setrange
/setmin
/setmax

7. GRAPHICS

7. GRAPHICS

7 Graphics

Graphics 7-1

Introduction 7-1

- Using OpenLookLabelGraphics 7-2

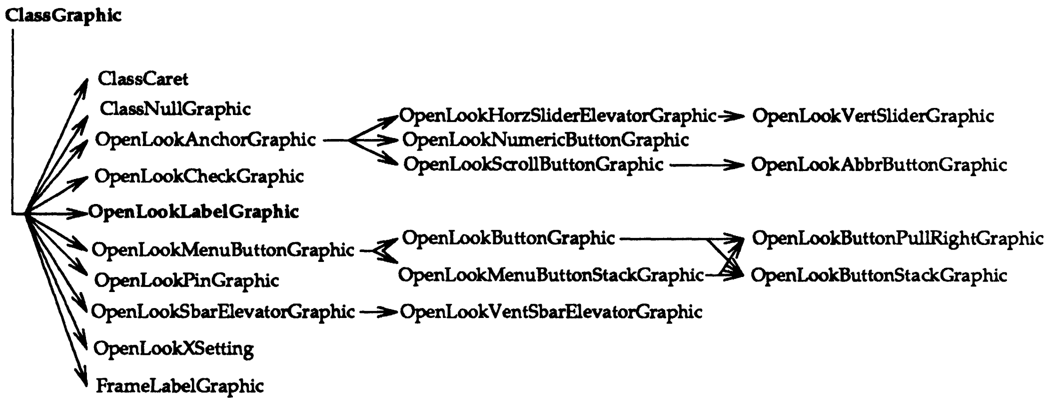
ClassGraphic 7-3

- State 7-3
- Size Negotiation 7-5
- Rendering Support 7-7
- Validation 7-10
- Building your own graphic 7-10

Examples 7-13

- Example 2: Complete code for the SimpleColorGraphic 7-15
- Example 3: OPEN LOOK compatible version of SimpleColorGraphic 7-17

Graphics



Introduction

Graphics are provided by the NeWS Development Environment as a packaging of a drawable object that knows how to reflect its state visually in an efficient manner. Graphics are designed to be very light weight so that they can be used as the images for buttons, menus, controls, and labels. There is no canvas associated with a graphic, so using a graphic does not imply the overhead of a NeWS canvas object or of a ClassCanvas object.

Another characteristic that makes graphics lighter weight than canvases is that they do not establish or maintain the graphics context. Any operation on a graphic, including **/setsize**, **/paint** or **/minsize**, is sensitive to the current canvas and transformation matrix and is likely to permanently alter the graphics context by changing the color or font. Users should be aware of this and wrap any calls to graphics in a **gsave/grestore** when appropriate. On the other hand, if a series of calls to graphics are going to be made, as in the case of menus, it is necessary to do only a single **gsave/grestore** pair for the whole series.

Using OpenLookLabelGraphics

Graphic instances are useful where a canvas interface is needed but a real instance of ClassCanvas is unnecessary. One example of this is labels on controls. By default the NeWS Development Environment controls do not come with labels but it is easy to construct a control with a label by packaging them together in a bag. There is a special subclass of ClassGraphic called OpenLookLabelGraphic that can be used just for this. Most common control labels are text so that is what we will use in this example. To create a label, you need to send **/new** to OpenLookLabelGraphic along with the thing you would like to display as the label. In this case it is the string "My Label:".

```
(My Label:) /new OpenLookLabelGraphic send
```

If you want to make your label use a different font than the default you can pass the font in with the string like this:

```
[(My Label:) /Helvetica findfont 12 scalefont]
/new OpenLookLabelGraphic send
```

You can also change the color of the label in a similar fashion.

```
[(My Label:) ColorDict /Green get] /new OpenLookLabelGraphic send
```

Now to complete the example we need to add the label to a bag along with a control. In this case I use an instance of an FlexBag to hold a label and a slider. I'll position the label to the left of the slider. Notice that I don't set the font or the color of the label; it will automatically inherit the text font and color from the bag.

```
/MyBag framebuffer /new FlexBag send def
/Label [/w {/w self POSITION}
        [ (My Label:) /Helvetica findfont 20 scalefont]
        /new OpenLookLabelGraphic send
        /minsize 1 index send /setsize 3 index send
        ] /addclient MyBag send
/Slider [/w {/e Previous POSITION}
        {pop} framebuffer /new OpenLookHorizontalSlider send
        0 0 300 20 /reshape 5 index send
        ] /addclient MyBag send

{} /setpaintproc MyBag send
10 10 /minsize MyBag send /reshape MyBag send
/activate MyBag send
/map MyBag send
/paint  MyBag send
```

ClassGraphic

The base class for all graphics is ClassGraphic. ClassGraphic is an abstract super class. This means that ClassGraphic can not instantiated directly, rather Class-Graphic must be subclassed and then the subclass instantiated. ClassGraphic packages a number of useful utilities that a subclasser or user of a graphic might be interested in using. It is not necessary to use all of the features that are packaged in ClassGraphic when building a subclass.

State

One of the most commonly used features of ClassGraphic is "state". The state transitions in a graphic are designed to be painted as efficiently as possible. For example, if a menu item is highlighted by drawing a box around it, the graphic should be able to draw that box without re-drawing the text. There are a number of methods can be used to efficiently manage the state of the graphic.

Methods:
/setinitstate
/setstate
/setelement
/state
/TranslateState
/EquivelantState?
/paint
/Fix

The states that a given graphic can reflect are completely up to the implementor of the graphic. There are two basics forms for the state supported by Class-Graphic, the state can be represented by any PostScript object type, or as an array of PostScript objects. The array case is provided as a convenience since it is a very common way of maintaining a number of axes of state (dependent or independent). Notice that since the graphic puts few restrictions on the form of the state, only graphics that understand the same states can be used inter-changeably. Graphic implementors are encouraged to describe the form of the state for their graphic above its definition. Objects that take graphics are also encouraged to publish the form of the state that they expect so that users know which types of graphics can be used with that object.

Each graphic class will respond in a unique way to changes in graphic state. Therefore, graphics implementors must override methods that directly interpret the meaning of the graphic state. Typically those methods are **/Fix** and **/paint**. **/Fix** is called when some element of the graphic state is changed by a call to **/setstate**. It takes as an argument a single boolean or an array of booleans, depending on how many independent states the graphic has. Each value in the array indicates a change in the corresponding state in the state array. **/Fix** should use these booleans to update only those graphic elements that have been directly afected by the change in state. For instance, one element of graphic state might indicate that the graphic should have a highlighted border. When that state element is changed the graphic should just update the border and not completely repaint itself. This will reduce the flashin the graphic as it changes state and increase the percieved speed of the update.

The **/paint** method also must concern itself with the state. **/paint** is called when the graphic needs to be drawn completely. The graphic implementor should make sure that their /paint method reflects the current state of the graphic when it is done.

Graphics implementors may also need to override **/EquivelantState?**. **/EquivelantState?** is called by **/setstate** to see if the new state is different from the old state (it is called for each element of the array when the array form is used). This method should return "true" if the old state and the new state differ, it should return "false" if the states are the same. The default implementation for **/EquivelantState?** is:

```
/EquivalentState? { % any any => bool
        % sufficient only for simple types.
        ne
} def
```

This will only work if the state elements are simple type (keywords, booleans, integers, etc.). If the state is represented by a more complex type (like a dictionary), then **/EquivalentState?** will need to be overridden compare the old state and the new state appropriately. In the case of a dictionary, it may be appropriate to compare the two dictionaries element by element. Notice that **/EquivalentState?** does not need to be overridden for state maintained as an array of simple types since it is called for each element of the array.

ClassGraphic does not initialize the state automatically, that is considered a subclasser's responsibility. The method **/setinitstate** should be called within **/newinit** to initialize the graphics state to one of its possible values.

Size Negotiation

ClassGraphic has a number of methods to support the manipulation of the size of the graphic as well as support for size negotiation. ClassGraphic is designed to work as a position independent object. That is that the graphic will always render at the "currentpoint" (defined in the process's NeWS graphics state). Some users of graphics find it convenient to treat the graphic like a ClassCanvas object for some operations (like positioning), so a number of ClassCanvas like position support methods exist. In general, it is recommended that these methods be avoided. The following list of methods can be used for size negotiation:

Size Methods:	Location Methods:
/minisize	**/move**
/preferredsize	**/location**
/setsize	**/reshape**
/size	**/paintat**

The **/minsize** method is used to determine the smallest size that the graphic can be made. ClassGraphic defines **/minsize** to return a width and height of zero, so graphic implementors should override **/minsize** to return the correct width and height for their graphics. ClassGraphic does not attempt to enforce the minimum size, rather it is expected that users of the graphic will call **/minsize** and respect the returned values. This is a performance consideration born of the fact that **/minsize** is typically a very complex function and should only be called when needed. The user of a graphic is in the best position to determine when they might violate the minimum size.

The **/preferredsize** size method is similar to **/minsize.** **/preferredsize** returns the width and height (larger than minsize) that the graphic can best be rendered at. For example, a scrollbar maybe able to be rendered so that some of its parts are not visible, it ideally it would like to be big enough to display all its parts and have some room to move. This size would be the scrollbar's preferred size. ClassGraphic defines **/preferredsize** to return the minimum size. A graphic implementor should override **/preferredsize** if the graphic has a preferred size different from the minimum size.

The size of the graphic can be set with the **/setsize** method. This method rarely needs to be overridden. There is no checking of the width and height arguments to ensure that they are larger than the minimum size. The user of the graphic is responsible for using reasonable arguments. **/setsize** has the side effect of invalidating the graphic (see validation below).

The **/size** method will return the current width and height of the graphic. If **/setsize** has not been previously called the graphic's minimum size will be returned.

The various location methods simple set, and retrieve the positional values that are the arguments (**/reshape** calls **/setsize** with the width and height arguments). These methods only perform a small subset of the functions of the similar methods in ClassCanvas. There use is discouraged.

Rendering Support

The various rendering support methods in ClassGraphic are also commonly used. Graphics have a number drawing methods in common with ClassCanvas. The following methods and class variables can be used to support the rendering of a graphic:

Methods:	Class Variables:
/paint	**/StrokeColor**
/Fix	**/TextColor**
/setthing	**/FillColor**
/thing	**/DisabledColor**
/thingatom	**/TextFamily**
/thingsequivalent?	**/TextSize**
/setterminal	**/TextEncoding**
/terminal?	**/TextFont**
/ThingSize	
/ShowThing	

ClassGraphic assumes that it is always rendered on a ClassCanvas instance. This allows the various variables above to inherit their values from the canvas that they are rendered on. These variables can be used at any time as arguments to PostScript operators (ex. setcolor, setfont) or methods. Graphics implementors can also choose to override the variables default behavior (getting their value from the canvas) by setting them to other values. Since these are well known names that users can change the defaults for in the UserProfile dictionary, graphic implementors are encouraged to use them.

The **/paint** and **/Fix** method have been discussed already. These are the two methods that get called to actually render the graphic. **/paint** should render the entire graphic including the elements of its current state. **/Fix** should do the most optimal job possible of painting the change in state indicated by its arguments. These methods should both render the graphic relative to the "currentpoint". When a user of a graphic calls either the **/paint** method or **/setstate** (**/setstate** will call **/Fix**) they should ensure that the current NeWS graphics state is correctly set up. The current point should be the location that the graphic is to be rendered at and the canvas should be the canvas that the graphic should be rendered on.

ClassGraphic supports a completely optional set of utilities for manipulating a "thing". These methods are supplied solely as an aid to graphic implementors that which to use them. A "thing" is generally used to hold the user supplied part of the graphic. The graphics that are used in tNt menus for menu command items, menu pullright items, and menu choice items all use ClassGraphic's thing support to hold the user supplied labels (usually strings). A thing is a simple specification that allows for a visible part, called the "atom" and some number of modifiers to affect the rendering of the atom. The thing specification looks like:

atom, or [atom modifier modifier ...]

An atom is either a string, executable array, instance of a graphic, or a canvas.

A modifier is any number of a font, a color, a pair of numbers, or a name.

The **/setthing** method can be called (typically in **/newinit**) to set the current "thing" for the graphic. The **/thing** method will return the current thing of the graphic. The **/thingatom** method will return the atom of the current thing of the graphic. **/thingsequivalent?** will compare its argument to see if it is equivalent to the current thing of the graphic. The two methods **/ThingSize** and **/ShowThing** know how to parse a thing to return its width and height or to show it at the current point. These methods should be called from methods like **/paint, /Fix** or **/minsize.**

The case where the atom is a string is the most common. The string is shown in the current font at the "currentpoint" such that the whole string is above and two the right of the current point when the CTM is the "defaultmatrix". The size of the thing is determined by the bounding box of the string.

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

```
{
        /paint eq {
                20 20 rect fill
        } {
                20 20
        } ifelse
}
```

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

The atom of thing can also be another graphic. The **/ThingSize** method will query the graphic for its size by calling the graphic's **/size** method, so the user should ensure that the correct size for the graphic has been set. The **/ShowThing** method will call the graphic's **/paint** method.

The canvas for of the atom has not been implemented as of this writing. A canvas can still be specified, but no bits will be rendered when **/ShowThing** is called. When **/ThingSize** is called on a thing with a canvas as its atom a width and height of zero will be returned.

Most of the modifiers for a thing change the current graphics state in some way. A modifier that is a font will set the current font to new font. A color modifier will change the current color. Numeric modifiers need to be specified as a pair, they cause the current point to be moved relatively by numeric pair (x then y). The final modifier type is the name of a method in the graphic that the thing is passed to. This method is executed every time either **/ThingSize** or **/ShowThing** is called. Here are a few example of valid things:

```
(test)
[ (red test) 1 0 0 rgbcolor ]
[ (offset red test) 1 0 0 rgbcolor 10 10 ]
[ (red times test) /TimesRoman findfont 12 scalefont 1 0 0 rgbcolor ]
```

The last two methods listed above, **/setterminal** and **/terminal?** are used by the graphic user to determine whether a given graphic should be treated as a "thing" or should be treated as a stand alone graphic. The **/terminal?** method is used by the the NeWS Development Environment menu code to determine if a graphic should be used as a thing to one of the menu's graphic types or treated as the complete menu item.

Validation

ClassGraphic has a number of methods that are used to track and modify the current validity of the graphic for painting.

The use of a **/ShowThing** is currently the only method of ClassGraphic that requires that an instance of its subclass be valid. Many of ClassGraphic's subclasses do take advantage of validation. The **/validate** method can be overridden to cache certain values in order to improve painting speed. One common use of validation is to cache the location of the "thing" relative to the current point for the current thing, size and font. The graphic is invalidated and then re-validated before the next time it is used. Deferring the validation until the time that the information is need allows for a number of operation that might invalidate the graphic to occur without having to validate the graphic each time.

Methods
/valid?
/validate
/?validate
/invalidate
/ValidateThing

Building your own graphic

Building a useful subclass of ClassGraphic is quite simple. The following example is a very simple graphic that has two states, either /Normal or /Highlighted. The graphic is a box that is filled a color that is specified as part of **/new** and is not outlined when the graphic is in its "/Normal" state and is outlined with a thick border when the graphic is in its "Highlighted" state. The initial state for this graphic will be "/Normal".

/newinit is overridden to consume the color argument to the graphic and to establish the initial state of the graphic. In this case, the state of the graphic is initialized by calling **/setinitstate** in **/newinit** with the name /Normal. Here is the code for /newinit:

```
% Override: initialize color & state.
/newinit { % color => -
        /newinit super send
        /GraphicColor exch def
        /Normal /setinitstate self send
} def
```

The second method that is overridden is **/paint**. Notice that **/paint** checks to see what the current state is and only paints the highlighting box if it is needed. See the complete example 1 for the details of the /PaintColorFrame and /PaintColorInterior utilities. Here is the code for **/paint** (notice that super is not called):

```
% Override: Paint color & border.
/paint { % - => -
        /state self send /Highlighted eq {
                HighlightThickness StrokeColor
                PaintColorFrame
        } if
        PaintColorInterior
} def
```

The **/Fix** method must also be completely overridden. **/Fix** will be called with a boolean that indicated whether the state has changed. The body of the /Fix method is wrapped in an "{ ... } if" so that no work is done if the argument to fix is "false". If the argument to **/Fix** is true then the state has changed and the new state needs to be reflected. In the code that follows the state is checked, if it is /Highlighted the the current color is set to the stoke color, if it is /Normal the current color is set to the fill color. Then the highlight frame is painted by the /PaintColorFrame utility (see Example 1 for the details of /PaintColor-Frame). Painting the highlight box in the fill color will have the effect or erasing just the highlight box. When fix is called the main color box is never re-drawn.

```
% Override: deal with state change
/Fix { % bool => -
        {
                /state self send /Highlighted eq {
                        HighlightThickness StrokeColor
                } {
                        HighlightThickness FillColor
                } ifelse
                PaintColorFrame
        } if
} def
```

The override of **/minsize** in this example is very simple, it simply returns a width and height of "30". This value is fairly arbitrary. However, the minsize does need to be at least big enough to accommodate the colored box and the highlighting box.

```
% Override: min size
/minsize { % - => width height
        30 30
} def
```

The final method that must be overridden is **/equivalent?**. This is the method that is called when a user wants to compare this graphic and some other graphic that they might have (for example, a search method in a menu). Since this method has no default implementation in ClassGraphic it must be provided by each subclass. **/equivalent?** needs to be carefully written since the graphic handed in may not understand the same methods as the graphic that is being implemented. In this case, another graphic is considered equivalent to this graphic if the color handed into **/new** is the same. The **/equivalent?** first checks that the graphic handed in as an argument understands the correct methods, and then checks that the colors are the same. If either of these tests fail, the **/equivalent?** returns false.

```
% Override: true if same GraphicColor.
/equivalent? { % graphic => bool
	/GraphicColor /understands? 2 index send {
		/GraphicColor exch send
		GraphicColor eq
	} {
		pop false
	} ifelse
} def
```

Methods:
/newinit
/paint
/Fix
/minsize
/equivalent?

Examples

Example 1 contains the complete code for the graphic just described. The following code can be entered in an interactive session to try the graphic:

```
/can framebuffer /new ClassCanvas send def
gsave
        5 5 scale
        160 130 60 40 /reshape can send
grestore
/activate can send
/map can send

false /settransparent can send

/g 1 0 0 rgbcolor
        /new SimpleColorGraphic send def

gsave
        can setcanvas
        5 5 moveto /paint g send
        5 5 moveto /Highlighted /setstate g send
        5 5 moveto /Normal /setstate g send
grestore

/destroy can send /can null def
/destroy g send /g null def
```

Example 2 contains a subclass of SimpleColorGraphic that supports the same states as the graphics required for the NeWS Development Environment menus and buttons. This graphic can be used in a tNt menu to build a color selection menu. The **/newinit** method for this graphic calls the **/setterminal** method with "true". This marks the graphic so that the menu or button will not try to treat it as a "thing", but will install it directly.

Example 2: Complete code for the SimpleColorGraphic

```
%
% State is: /Normal or /Highlighted
%
/SimpleColorGraphic ClassGraphic
[/GraphicColor] classbegin

% Class Variables:

        /HighlightThickness        2 def
        /Inset HighlightThickness 1 add def

% Methods:

        % Override: initialize color & state.
        /newinit { % color => -
                /newinit super send
                /GraphicColor exch def
                /Normal /setinitstate self send
        } def

        % Override: Paint color & border.
        /paint { % - => -
                /state self send /Highlighted eq {
                        HighlightThickness StrokeColor
                        PaintColorFrame
                } if
                PaintColorInterior
        } def

        % Util: Paint border; preserve currentpoint
        /PaintColorFrame { % thickness color => -
                setcolor setlinewidth
                currentpoint
                /size self send rect stroke
                moveto
        } def

        % Utility: Paint interior
        /PaintColorInterior { % thickness color => -
                Inset dup rmoveto
                /size self send
                Inset 2 mul sub exch Inset 2 mul sub exch
                rect
                GraphicColor setcolor fill
```

(continued on next page)

```
		} def

		% Override: deal with state change
		/Fix { % bool => -
			{
				/state self send /Highlighted eq {
					HighlightThickness StrokeColor
				} {
					HighlightThickness FillColor
				} ifelse
				PaintColorFrame
			} if
		} def

		% Override: min size
		/minsize { % - => width height
			30 30
		} def

		% Override: true if same GraphicColor.
		/equivalent? { % graphic => bool
			/GraphicColor /understands? 2 index send{
				/GraphicColor exch send
				GraphicColor eq
			} {
				pop false
			} ifelse
		} def
classend def
```

Example 3: OPEN LOOK compatible version of SimpleColorGraphic

```
% Modify SimpleColorGraphic to handle OpenLook
% three element state array:
%       [       /Highlighted|/Normal
%               /Enabled|/Disabled
%               /Emphasized|/Normal
%       ]
/OpenLookColorGraphic SimpleColorGraphic []
classbegin
% Class Variables:
        /EmphasizeThickness     1 def

% Methods:
                % Override: Adjust for OL-ness.
        /newinit { % color => -
                /newinit super send
                [/Normal /Enabled /Normal]
                /setinitstate self send
                true /setterminal self send
        } def

        % Override: Handle more complex state.
        /paint { % - => -
                /state self send
                dup 0 get /Highlighted eq {
                        HighlightThickness StrokeColor
                        PaintColorFrame
                } {
                        dup 2 get /Emphasized eq {
                                EmphasizeThickness StrokeColor
                                PaintColorFrame
                        } if
                } ifelse
                pop
                PaintColorInterior
        } def

        % Override: handle more complex state
        /Fix { % [bool bool bool] => -
                0 exch {
                        {
                                /state self send 1 index get
                                1 index /PaintState self send
```

(continued on next page)

```
                } if
                1 add
        } forall
        pop
} def

% Utility: Repaint changed state.
/PaintState { % name which => -
        {
        0 { % /Highlighted or /Normal
                /Highlighted eq {
                        HighlightThickness StrokeColor
                        PaintColorFrame
                } {
                        HighlightThickness FillColor
                        PaintColorFrame
                        State 2 get /Emphasized eq {
                                EmphasizeThickness StrokeColor
                                        PaintColorFrame
                        } if
                } ifelse
        }
        1 {pop} % /Enabled or /Disabled; ignore!
        2 { % /Emphasized or /Normal
                State 0 get /Highlighted eq {pop} {
                        /Emphasized eq {
                                StrokeColor
                        } {
                                FillColor
                        } ifelse
                        EmphasizeThickness exch
                        PaintColorFrame
                } ifelse
        }
        } case
} def
```

8. NeWS DEVELOPMENT ENVIRONMENT

8. NeWS DEVELOPMENT ENVIRONMENT

8 The NeWS Development Environment Input Model

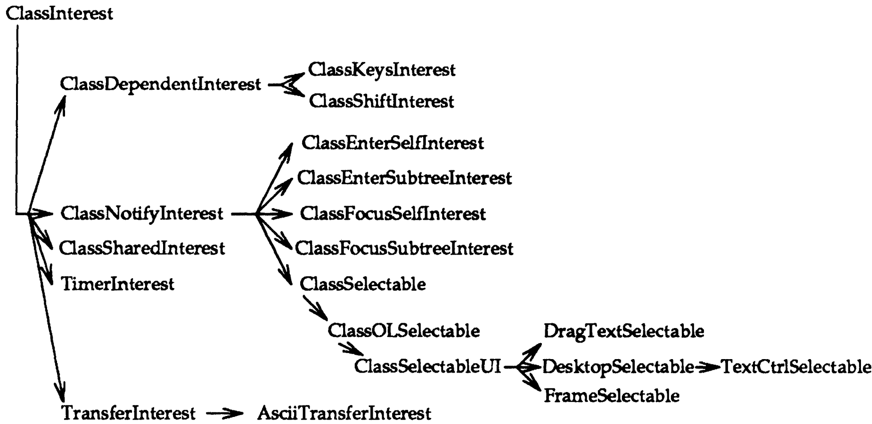
The NeWS Development Environment Input Model 8-1
Introduction 8-1
Review of NeWS Input 8-2
Executable Matches 8-3
Main Class Hierarchies 8-5
- ClassEventMgr 8-5
- ClassKeyboard 8-5
- ClassInterest 8-5

Branch Hierarchies 8-6
- ClassNotifyInterest 8-6
- ClassDependentInterest 8-7
- Keyboard Processing 8-7

Methods 8-9
- ClassInterest 8-9
- ClassNotifyInterest 8-10
- ClassDependentInterest 8-11
- ClassShiftInterest 8-12
- ClassKeysInterest 8-12

Examples 8-14
- Example 1: Simplest Keyboard Processing 8-14
- Example 2: Adding Function Keys 8-15
- Underlying Mechanisms of Examples 1 and 2 8-17
- Example 3: Reading the Number Pad 8-19
- Example 4: Non-Standard Uses 8-21
- Complex Example 8-23

The NeWS Development Environment Input Model



Introduction

The basic NeWS input system, as described in the *NeWS Programmer's Guide,* provides a complete mechanism for handling user inputs in PostScript applications. However, it is defined at a fairly low level; the amount of detailed knowledge required to build applications at this level is daunting. Further, it has no relation to the class system in which the NeWS Development Environment is built; this leads to inconsistencies and conflicts in the structure of applications if both are used.

A "class-based" layer of facilities has been built on top of the fundamental input and process primitives. This encapsulates many common forms of processing, and has a uniform style with the rest of the NeWS Development Environment class system. This layer comprises class hierarchies under three base classes: ClassEventMgr, ClassKeyboard, and ClassInterest. ClassEventMgr provides a process which expresses interests, awaits events which match those interests, and then dispatches to client handlers for those events. ClassKeyboard is a utility class which provides the definition of the keyboard attached to the server, along with a number of methods for inquiring and manipulating aspects of that definition. ClassInterest itself provides a fairly thin veneer on the interests

defined by the server; it has a rich tree of subclasses which provide for a great deal of common input processing such as keyboard input and selections.

The facilities described in this section are, in turn, used by higher-level components of the NeWS Development Environment. For instance, many applications which require keyboard input can get it conveniently via the OpenLookTextControl. Even where the NeWS Development Environment does not provide all the desired functionality, it is unlikely that the lowest-level facilities are the appropriate ones to use. The first few examples described in this chapter show ways to get straightforward keyboard input; check them before deciding whether you should master all the intervening material.

For the simplest clients, there is no need to use an instance of ClassInterest. Interests returned by the MakeInterest utility in ClassCanvas are equally acceptable to an event manager. This level of use might be appropriate, for instance, for detecting button hits on a canvas, without reference to focus, selections, etc.

Review of NeWS Input

Let's briefly review the underlying server semantics for input. An input event is one of the "magic dictionary" objects (type **/eventtype**). It has a number of fields defined by the server, including Name, Action, Canvas, Process, KeyState, and TimeStamp; clients are free to add other fields just as though the event were a standard dictionary. Input events are generated by the server in response to hardware and window system activity (key presses, window crossings, etc.). Events are also generated by client processes running in NeWS. The server distributes an event by matching it against *interests* expressed by processes, where an interest is simply another event which has some of its Name, Action, and Canvas fields filled in with desired values. Interests may be directed at sets of values (rather than single values) by storing compound objects (arrays or dictionaries) in the relevant fields.

The *executable match* facility is available and mentioned in the server documentation, but receives relatively little emphasis. This is a central component of the NeWS Development Environment's subclasses of ClassInterest, so we will describe it in some detail here.

Executable Matches

When a dictionary appears in the Name, Action, or Canvas field of an interest, it specifies that an event will match any of a number of values (the keys in the dictionary). What happens when such a match is found depends on the value associated with the key in the dictionary. If the dictionary value is not executable, it replaces the value in the event being delivered. This form can be used, for example, to translate a key code into a character. But if the dictionary value is executable, the value in the field is not modified; instead the executable value is evaluated in the context of the awaitevent which receives the event. If more than one of those three fields has an executable match, all are evaluated; the order is Name, then Action, then Canvas.

Let's consider a simplified keyboard example. An event generated in response to a key press on the 'A' key will have a keycode in its Name field (e.g., 28493); its Action will be **/DownTransition**, and it will have null in its Canvas field.

First consider how this would be handled without executable matches. The client might use an interest defined with

```
Name:     dict[ ...
          28493: 97        % the keycode defined to ASCII 'a'
          ... ]

Action:   /DownTransition

Canvas:   null
```

That interest would be expressed in a process that then executed some code like

```
{awaitevent /KeyDown mycanvas send} loop
```

The matched events will be returned by the awaitevent (with the keycode translated to a character); the client's KeyDown method consumes that event, and then the loop returns to the awaitevent.

An equivalent formulation using executable matches would use an interest like

```
Name:        dict[ ...
             28493: 97      % the keycode defined to ASCII 'a'
             ... ]

Action:      dict[ /DownTransition: {/KeyDown //mycanvas send} ]

Canvas:      null
```

and the event-processing loop is simply

```
{awaitevent} loop
```

The translation from keycode to character happens the same way in both cases; but by convention an executable match consumes matched events. If there is more than one executable match in a single interest, the last (and *only* the last) should consume the event. It is important that there be no mixing of interests which use executable matches in the same process with those that don't; the stack discipline requires that a process use one or the other uniformly.

The executable match style has some advantage in expressive clarity: the way a particular value is handled is closely associated with that value. It also ensures the system can provide handlers for events it needs to process, while allowing the client to add other events to be handled in the same process. For these reasons, all instances of ClassEventMgr require executable-match interests.

Main Class Hierarchies

Now let's return to the three main classes involved in processing user inputs.

ClassEventMgr

ClassEventMgr is the NeWS Development Environment analogue of the forkeventmgr and ExpressEmgrInterests utilities. It provides for starting a process which will express a set of interests (**/new, /clearcontext, /setname**); adding and removing interests from the set managed by that process (**/addclient, /addclients, /removeclient, /removeclients**); and shutting down the event manager process (**/destroy, /queuedestroy, /removeallclients**).

It also provides a set of inquiry methods (**/name, /active?, /getprocess, /interests, /processstate**), mechanisms for evaluating executable code in the event manager process (**/callmanager, /argcallmanager**), and support for a *robust* form of eventmgr, which is not killed by errors occurring in handlers for the events it is receiving (**/makerobust**). As mentioned above, any interest passed to an instance of ClassEventMgr must specify an executable match.

ClassKeyboard

ClassKeyboard provides the description of the server's keyboard — what keys are available on it, where the characters and modifier keys are, what escape sequences are associated with which function keys, etc. It also provides a number of utility methods to clients interested in keyboard processing; the most useful are **/toChar, /toControl, /toControlChar, /toLower, /toMeta, /keyforsymbol, /buildkeydict**, and the pair **/removefunctionkey** and **/restorefunctionkey**.

ClassInterest

ClassInterest and its subclasses are the NeWS Development Environment analogue of the eventmgrinterest utility; they are the workhorses of input processing. It is possible to instantiate ClassInterest itself; the result is an interest little different from that returned by createevent. Rather, most clients deal with subclasses of this class (both their own, and a few system-defined subclasses).

Branch Hierarchies

There are two main branches to the class tree under ClassInterest: ClassNotifyInterest and ClassDependentInterest. They often occur in related groups. Both of these subclasses require that their instances specify executable matches (unlike ClassInterest itself, which may or may not, at the client's convenience).

ClassNotifyInterest

A Notify interest matches an event which acts as a trigger or initiates a state. The two most common examples are assignment of focus to a particular canvas (ClassFocusSelfInterest) and detection of a mouse-button down or other event which initiates a selection-making dialogue (ClassSelectable). The Notify interest may also detect the event which terminates the state it initiated; ClassFocusSelfInterest does, but ClassSelectable does not. A Notify interest is normally handed to an eventmanager shortly after it is created, and remains active thereafter for the life of its client.

Three other subclasses of ClassNotifyInterest are less frequently used, but are provided for completeness. ClassFocusSubtreeInterest triggers when focus is assigned to a canvas or any of its descendants in the window hierarchy. ClassEnterSelfInterest detects entry of the cursor directly into a particular canvas (whether or not this causes the canvas to become the input focus). ClassEnterSubtreeInterest detects entry of the cursor into a canvas or any of its descendants.

ClassFocusSelfInterest has one additional side effect: when it is activated, it establishes its canvas as a focus client. That is, it tells the global focus manager that the canvas is interested in being sent focus events. If the canvas is a ClassCanvas, the FocusSelfInterest calls **/setkeyconsumer** (which in turn calls addfocusclient); otherwise it calls addfocusclient directly. The FocusSelfInterest likewise takes care of removing the canvas as a focus client when the interest is deactivated. Note that the other three subclasses do *not* register the canvas as a focus client.

ClassDependentInterest

When a Notify interest is triggered (i.e., when an event arrives which matches the interest), one of the things the handler does is to express other interests which should only be active for the duration of the state initiated by the Notify interest. These transient interests MUST in turn be instances of some subclass of ClassDependentInterest.

For a focus Notify interest, the dependent interests concern keyboard events, on both shift keys and those that directly produce characters. (These will be instances of ClassShiftInterest and ClassKeysInterest, respectively.)

Dependents of a Selectable will handle mouse motion or crossing events, and button-up on the triggering button; these dependents are less structured than in the keyboard realm.

Generally, a DependentInterest will be global (i.e., it will be a pre-child interest on the root canvas), while a Notify interest will be expressed on a particular canvas.

Keyboard Processing

Keyboard processing is normally handled by instances of ClassFocusSelfInterest, ClassShiftInterest, and ClassKeysInterest. Straightforward keyboard clients are provided for in methods which hide almost all of the interrelations of these classes; see the first examples below. (For these purposes, a "straightforward keyboard client" is one which requires only ASCII characters off the standard typing array of the keyboard, in the usual shift combinations. By default, the usual shifts are Control, Shift, Meta, and CapsLock; Meta may be excluded from that set by defining **/MetaKeys?** to false in the UserProfile dict at startup.) More sophisticated use involves deeper understanding of these classes, and is addressed in succeeding sections.

There are two main sources of complication in handling keyboard input:

The client should not get keyboard events when it is not the focus. However, when it becomes the focus, it should get events regardless of the cursor location. (Click-to-type will not work unless this criterion is met.)

The interpretation of keys varies with the shift state, but different clients and different keyboards may require different definitions of which keys determine the shift state. For instance, emacs will almost certainly use the Meta keys, but other editors may apply other interpretations to those keys.

Similarly, some keyboards have a NumLock key which should change the right pad to be a numeric key-pad, but should not affect keys in the standard typing array.

A brief sketch of how keyboard events are delivered to clients will indicate how these problems are addressed.

An instance of ClassFocusSelfInterest detects when the client's canvas has been assigned focus; until that happens, the client is uninterested in keyboard events. When the client gets focus, the focus Notify interest activates its associated Shift interests and Keys interests. A Shift interest determines the state of interesting shift keys when it is activated, and tracks up- and down- transitions on its shift keys to maintain that state as long as it is active. A Keys interest generally notices down-transitions on standard keys, translates them to characters, and delivers them to the client. The translation performed by a Keys interest is determined by the dictionary in its Name; when a Shift interest recognizes a change in the shift state, it causes new dictionaries to be stored in the Names of its associated Keys interests. When the client canvas loses focus, the dependent interests (Shift interests and Keys interests) are deactivated by the focus Notify interest.

This second problem is addressed by the relationship of Shift interests and Keys interests. A Keys interest has an **/update** method that causes the interest to modify itself based on an externally provided state. The default behavior is that **/update** takes an integer and modifies the Name field of the interest based on that integer, as follows.

When a Shift interest is activated, and again whenever the state of the shift keys changes, it reports the shift state to each of its associated Keys interests. Each Keys interest uses the shift state to index into an array of dicts, and stores the selected dict into its Name field. The array of dicts is precomputed so that the dict corresponding to a particular shift state will map keystations into ASCII characters according to that shift state. The result is that most keystrokes are converted directly into ASCII characters without having to examine the shift state on each keystroke; the additional work for handling shift keys is done only when the shift keys themselves change state.

Multiple Keys interests may be associated with a single Shift interest, and multiple Shift interests may be associated with a single Notify interest. Methods are provided in each class for inserting and removing interests at each location in the hierarchy.

It is also possible for a Dependent interest to be activated independently, despite its name. This is primarily useful for a global Shift interest which maintains a fully-defined shift state, without any associated Keys interests, for inquiries by random clients.

Methods

This section summarizes the interesting methods of these various classes. Methods for ClassKeyboard are discussed in a separate section. Two other data structures are crucial to the understanding of keyboard processing in the subclasses of ClassInterest: the ShiftDict and Map.

A ShiftDict is a dictionary that specifies how a shift state is computed from a set of keys; it associates device-keycodes to a bit in the shift state, or to some procedure that does more complicated state maintenance. This is a little tricky; if you're going to build your own ShiftDict, see the full description below.

A Map is a dictionary that carries some set of keycodes to values such as characters. At any time a Keys interest is active, its Name field will contain such a map.

Each Keys interest has an associated set of Maps: an array of 2^n dicts, where n is the number of bits in the shift state. If a shift-key transition is detected, the map corresponding to the new shift state is stored in the Name field; this is how different values are reported for the same key depending on shift state.

ClassInterest

`/new`

`canvas action name` **/new** `interest`
Create a new instance of the class.

`/interest`

`--` **/interest** `interest`
Analogous to **/canvas** in ClassCanvas; obtains the NeWS interest that corresponds to the ClassInterest. (Currently this is "self", but clients may use this method to isolate themselves in case the implementation changes.)

`/activate` `event` **/activate** `--`

Express the interest. Redundant (but safe) if already expressed. If the Process field in the interest is non-null, the interest is expressed for that process; otherwise it is expressed for the current process. The event argument is present because some subclasses require it, and callers may not know whether this particular interest belongs to such a class. If no useful event is available to hand to **/activate**, use nullevent.

`/deactivate` `--` **/deactivate** `--`

Revoke the interest. No-op if the interest has not been expressed.

`/Active?` `--` **/Active?** `bool`

Has this interest been activated (expressed)?

`/destroy` `--` **/destroy** `--`

Automatically nulls out the Name, Action, and Canvas fields, to facilitate garbage collection.

ClassNotifyInterest

`/NotifyIn` `event` **/NotifyIn** `--`

Activate all Dependent interests of this Notify interest. Generally called as part of an executable match in the Notify interest, so the triggering event is on the stack.

`/NotifyOut` `event` **/NotifyOut** `--`

Deactivate all Dependents. No-op if they have not been activated.

`/TestTrigger` `--` **/TestTrigger** `bool`

Tests whether the triggering event for this interest has already occurred. This is used when the Notify itself is activated, when it wants to determine if it should immediately activate its

Dependents. The default implementation for **/TestTrigger** is to return 'false'; it is overridden by subclassers.

`/finddependent`

`value` **/finddependent** `Dependent true/false`

Find the Dependent interest (if any) registered using the given value. (See **/adddependency** below.) The **/addsuite** method in a Keys interest uses this method to avoid creating two identical Shift interests.

ClassDependentInterest

`/new`

`any Nint canvas action name /new Dint`

`/new`

`null canvas action name /new Dint`

If a non-null Notify interest is provided, the Dependent interest is automatically registered with the Nint, in which case the "any" argument must be provided for use with **/adddependency**.

`/adddependency`

`any Nint` **/adddependency** --

Make this interest dependent on a specified Notify interest. The Dependent is first disconnected from its current Notify, if any. The "any" is a value that can later be used to obtain the Dependent from the Notify using **/finddependent**. If this Dependent is a Shift interest, it calls **/adddependency** for its associated Keys interests so that they will all be activated by the same Notify interest.

`/removedependency`

-- **/removedependency** --

Disconnect this interest from its current Notify, if any. If this interest is a Shift interest, it calls **/removedependency** for its Keys interests as well.

ClassShiftInterest

`/new`

`shiftdict Nint` **/new** `Sint`

This method is not often called by clients; instead, Shift interests are created by calling **/addsuite** for a ClassKeysInterest (below). The **/Action** and **/Canvas** fields for a Shift interest are hard-wired. The shiftdict gets stored in the **/Name** field; if the dict is null, the default shifts (which may or may not include Meta, depending on **/MetaKeys?** in UserProfile) are used. The Notify interest can be null to create a global Shift interest. If it is non-null, the shiftdict is used as the "any" for **/adddependency**.

`/destroyempty`

`--` **/destroyempty** `--`

Destroy this Shift interest if it has no associated Keys interests.

`/shiftstate`

`--` **/shiftstate** `int`

Return the current shift state of this interest.

`/modifierdown?`

`name` **/modifierdown?** `bool`

Return whether a given modifier is down, according to this interest. The argument is a name evaluated by sending it to the interest. Usually the name is a class variable. ClassShiftInterest defines class variables for these common shift names: **/Shift, /Caps, /Control, /Meta.**

ClassKeysInterest

`/new`

`downproc upproc maps shiftdict` **/new** `interest`

The usual interest fields are hard-wired. Name is initially null, to be updated when we hear from the Shift interest; Canvas is null because Keys interests are always global, etc. The client-supplied parameters are instead a proc to be

called when a key is seen to go down (null if none), another proc for keys going up, an array of 2^n keymap dicts, and a shiftdict to be used when building the Shift interest. The keymaps and shiftdict can be null to get the corresponding defaults. Note that unwanted downprocs and upprocs should be null, not nullproc, for maximum efficiency.

`/addshift`

`Sint` **/addshift** --

Link this interest with a particular Shift interest. The Keys is first disconnected from its current Shift, if any. The Shift tells you which Notify to hook up to (by calling **/adddependency**).

`/removeshift`

-- **/removeshift** --

Disconnect this interest from its current Shift, if any.

`/update`

`int` **/update** --

Update this interest based on the Shift's new state. The Keys interest selects the specified dict from the "maps" array supplied to **/new**, and stores the dictionary in the interest's Name field.

`/shiftstate`

-- **/shiftstate** `int`

Return the current shift state of the associated Shift interest. This method is provided for clients who let the Keys interest create the Shift (using the methods below), so the client has no handle by which to ask the Shift directly.

`/addsuite`

`notify-int` **/addsuite** --

Add a Keys interest to the structure under a given Notify interest. If the Notify already has a Shift interest that uses the shift dict given to the Keys' **/new** method, the Keys is added to that Shift. Otherwise a new Shift interest is created and added to the Notify, and the Keys is added to the new Shift.

`/removesuite`

-- **/removesuite** --

Unlink a Keys interest from its Notify/Shift structure. If this leaves the Shift with no Keys, the Shift is destroyed.

`/defaultkeys`

`downproc canvas` **/defaultkeys** `Nint`

This creates a Notify interest in input focus on the given canvas, and also a Keys interest with the given downproc and no upproc, using the default keymaps and shift dict. The Keys' **/addsuite** is called to create the Shift interest and hook everything together, and the NOTIFY interest (not the Keys) is returned. Thus, /defaultkeys is often called in a ClassCanvas's **/MakeInterests** method.

`/metakeys`

`downproc canvas` **/metakeys** `Nint`

Same as **/defaultkeys**, except it always includes the Meta shift keys rather than using the default specified via the User Profile **/MetaKeys?** boolean.

Examples

Example 1: Simplest Keyboard Processing

A client which needs only ASCII characters from the standard typing array of the keyboard has very little to do; it simply creates one more interest for its frame or canvas event manager to manage. This example, which uses the ReportChar procedure, extracts a character from the Name of an event and prints it on the current file.

```
/ReportChar { % event => -
        /Name get =
} def

/MakeInterests {
        /MakeInterests super send
        /ReportChar self soften buildsend
        Canvas /defaultkeys ClassKeysInterest send
} def
```

The method **/defaultkeys** returns a single focus Notify interest. As a side effect of expressing that interest, the client's canvas is declared a focus client; that is, the global focus manager is told that the focus should be assigned to that canvas under the appropriate circumstances. Now, when focus is assigned to the client canvas, the Notify interest will activate default Shift and Keys interests. The first will track the state of the shift keys; the latter will translate keycodes to characters according to the current shift-state, and invoke ReportChar to handle those characters. ReportChar will be called with an event on the stack, which it consumes.

The underlying mechanisms for all this are discussed below.

Example 2: Adding Function Keys

Suppose a client wants to receive function-key events, reported by the name of the key, as well as the simple ASCII characters. Let's re-write the first example to do this.

```
/MakeInterests {
        /MakeInterests super send

        % As before, use /defaultkeys to create a focus Notify interest,
        % and include a simple ascii interest dependent on it.
        /ReportChar self soften buildsend
                Canvas /defaultkeys ClassKeysInterest send   % Ni

        % Now create another Keys interest, in function keys,
        % and add it to the same Notify interest.
        dup                                             % Ni Ni
        /ReportFunction self soften buildsend           % Ni Ni down-proc
        null                                            % Ni Ni dn up
        [ /FKeyNames ClassKeyboard send                 % Ni Ni dn up [ shiftdict
        /DefaultKeyDicts ClassKeysInterest send         % Ni Ni dn up [ sd defmaps
                length 1 sub {dup} repeat ]             % Ni Ni dn up maps
        null /new ClassKeysInterest send                % Ni Ni FKey-interest
        /addsuite exch send                             % Ni
} def
```

The client now is using two private methods, **/ReportChar** and **/ReportFunction**, to respond to keyboard events. **/ReportChar** handles the default keys, while a second Keys interest is constructed to catch function-key events, and handle them with **/ReportFunction**. In both cases, the client wants only to see the key-presses; key-releases are uninteresting. The method **/defaultkeys** gives this behavior automatically. A general Keys interest, as created for the function keys, allows treating up and down transitions independently; passing null in place of a handler for either transition causes it to be ignored.

We want this interest to share the default-keys interest's shift dict; but the shift state should not affect the reported values of the function keys. Therefore we construct a keymaps array by simply making enough copies of the same map. This map is the **/FKeyNames** dict supplied by ClassKeyboard, which maps key-codes to function-key names (like **/FunctionF1**); "enough" is determined by matching the size of the default keymaps array. The shift dict is "null", which results in this Keys interest using the same shift dict as **/defaultkeys**.

Having created this second Keys interest, we tell it to insert itself as another dependent of the Notify interest returned by **/defaultkeys**; and MakeInterests returns just the Notify interest. Processing by the event manager is just as in Example 1.

Underlying Mechanisms of Examples 1 and 2

The method **/defaultkeys** performs the following magic:

1. A Notify interest in focus directly in the client's canvas is created:

```
can /new ClassFocusSelfInterest send
```

   This is special to **/defaultkeys** and **/metakeys** which are ordinary Keys. Interests do not create their own Notify interests.

2. A Keys interest is created using the default keymaps, and a dict is created when a DownTransition is received on any of the specified keys.
3. A Shift interest in the default set of shift keys is created.

The three interests are chained together, and the Notify interest is returned. When the ClassFocusSelfInterest is activated, it in turn declares the client canvas (can) to be a focus client:

```
can addfocusclient  -or-  true /setkeyconsumer can send
```

(The latter is used if the canvas is a ClassCanvas. The method **/setkeyconsumer** in turn calls addfocusclient.) ClassFocusSelfInterest also calls removefocusclient (or false **/setkeyconsumer**) when deactivated.

Sometime later, the user performs an action that assigns the focus to can (e.g., clicks in the window). A focus notification event is sent, and it matches the FocusSelfInterest.

Executable match code in the FocusSelfInterest loops over its Dependents, sending **/activate** with the notification event to each. (In this case, there are two: the anonymous Keys interest and Shift interest created by **/defaultkeys.**)

The Shift interest's **/activate** method computes its shift state based on the event's KeyState, expresses the Shift interest, and sends **/update** with the shift state to each of its Keys interests. (There is only one of these in the first example: the one created by **/defaultkeys**. The second example also has the function-key interest attached to the same Shift interest.)

The Keys interest's **/update** method uses the shift state to select an appropriate dict from its Maps and stores that in its Name field.

Now a key-down on any of the standard keys will match the Keys interest; the dict in the Name field will translate the key code to the appropriate character (in the case selected by the current shift state), and the NeWS event mechanism will automatically store the translated value back into the event (NOT the interest). The dict in the Action field then causes the client's callback to be called with the translated event on the stack.

If a shift-key transition comes through while these interests are active, executable-match code in the Shift interest recomputes its shift state and sends the new value with **/update** to the Keys, thus updating the translation dict in its Name field.

The function-key interest is another Keys interest, which arrives at a similar object via a more explicit specification.

The keymap that maps keycodes to names has already been computed by ClassKeyboard, so we simply share that. If we had desired instead to get the appropriate escape sequences, like "^[[224z" for the F1 key, we could have used a parallel map, FKeyStrings.

The Maps field of the interest is set to an array which consists of 16 copies of this dict (or 8, if MetaKeys? is false). Thus, this dict will be stored in the Name of the interest regardless of the state of the shift keys. The Action field is set to be a dict in which **/DownTransition** is defined to a proc that sends **/ReportFunction** to the canvas. If a proc had been passed in place of the null argument to **/new**, **/UpTransition** would be defined to that proc; but as it is, there is only one entry in the dict.

This Keys interest is associated with the same Notify and Shift interests as the first; it is activated, updated, and deactivated at the same time and with the same arguments.

Example 3: Reading the Number Pad

The keyboard has a Num Lock key on the right keypad, which affects all the right-pad keys below the top row. The lock should be treated like Caps; one press sets it, the next clears it. (Also like Caps, it's a matter of taste whether it should take effect globally or on a per-window basis.) One way to get this is to define a shift state which has only 1 bit, and to track the Num Lock key exactly as the Caps key is tracked in the ShiftInterest code:

```
/NumShiftDict 1 dict dup begin
        /NumLock /keyforsymbol ClassKeyboard send
                [1 /LockChange] buildinterestsend def
end def
```

Next we define a keymap with 17 entries, for the 17 keys on the right pad that should be affected by the Num Lock key:

```
/NumKeysDict 20 dict dup
        {       /NumPadEqual keyforsymbol    (=) toChar
                /NumPadSlash keyforsymbol    (/) toChar
                /NumPadStar keyforsymbol     (*) toChar
                /NumPadMinus keyforsymbol    (-) toChar
                /NumPad7 keyforsymbol                 (7) toChar
                /NumPad8 keyforsymbol                 (8) toChar
                /NumPad9 keyforsymbol                 (9) toChar
                /NumPadPlus keyforsymbol     (+) toChar
                /NumPad4 keyforsymbol                 (4) toChar
                /NumPad5 keyforsymbol                 (5) toChar
                /NumPad6 keyforsymbol                 (6) toChar
                /NumPad1 keyforsymbol                 (1) toChar
                /NumPad2 keyforsymbol                 (2) toChar
                /NumPad3 keyforsymbol                 (3) toChar
                /NumPadEnter keyforsymbol    (0 toChar
                /NumPad0 keyforsymbol                 (0) toChar
                /NumPadDot keyforsymbol               (.) toChar
        } /buildkeydict ClassKeyboard send
def
```

The **/buildkeydict** method executes the {...} proc in the context of ClassKeyboard, so keyforsymbol and toChar can be invoked directly. The {...} puts keycode/value pairs on the stack, and **/buildkeydict** then defines those pairs into the given dict.

The other keymap, for use when NumLock is not set, has no entries, so we'll use nulldict in that position of the Maps array:

```
/NumLockMaps        [nulldict NumKeysDict] def
```

Now, the following code would go in the canvas' MakeInterests:

```
/ReportChar self soften buildsend
        null NumLockMaps NumShiftDict
        /new ClassKeysInterest send
dup /Priority 1 put
/addsuite exch send
```

The sense of all this is, "when the NumLock is set, accept and translate those 17 keys to the characters given on key-down; otherwise ignore everything." Raising the priority prevents those keys from also being interpreted as function keys, in the function-key interest created earlier.

Note that this Keys interest has a different Shift interest than the two defined above, although they are dependent on the same Notify interest. An alternative approach would have used a single shift dict (with 5 bits of state). However, this would have have required larger Maps arrays all around, and construction of a larger shift dict to cover all the possibilities. Either approach would be reasonable.

Example 4: Non-Standard Uses

Now let's look at a simple example of non-standard usage. Suppose you want to have a pinball game that uses the shift keys to control the flippers. Thus you want to be notified of both down and up transitions on the shift keys, and don't care about any other keys. Moreover, you want to be notified directly about the shift keys, rather than having them modify the treatment of other keys.

Here is the complete PostScript code for implementing the canvas, suitable for psh'ing. The discussion that follows will center on the keyboard-related aspects.

```
% flippers.ps: creates a canvas that (when it has the focus) flips flippers
% based on the state of the shift keys.

/FlipperCanvas ClassCanvas dictbegin
        /Left false def
        /Right false def
dictend
classbegin

        % Give this canvas a 0-1 0-1 coordinate system.
        /Transform { % x y w h => x' y' w' h
                4 2 roll                % w h x y
                translate               % w h
                scale                   % -
                0 0 1 1
        } def

        /PaintCanvas {
                FillColor /FillCanvas self send
                .2 .5 moveto .25 Left .1 -.1 ifelse rlineto
                .8 .5 moveto -.25 Right .1 -.1 ifelse rlineto
                .025 setlinewidth StrokeColor setcolor stroke
        } def

        /KeyMap dictbegin
                /LeftShift /keyforsymbol ClassKeyboard send     /Left    def
                /RightShift /keyforsymbol ClassKeyboard send    /Right   def
        dictend def

        /MakeInterests {
                /MakeInterests super send

                self soften /new ClassFocusSelfInterest send    % Nint
                dup
```

(continued on next page)

```
                [true /KeyEvent] self soften buildsend
                [false /KeyEvent] self soften buildsend
                [KeyMap] nulldict /new ClassKeysInterest send  % Nint Nint Kint
                dup /Exclusivity false put
                /addsuite exch send                            % Nint
        } def

        /KeyEvent {         % event bool => -
                exch /Name get exch def
                /paint self send
        } def

classend def

/fc FlipperCanvas nullarray framebuffer /newdefault ClassBaseFrame send def

(Flippers) /setlabel fc send
/activate fc send
/reshapefromuser fc send
/map fc send

newprocessgroup
currentfile closefile
```

All the keyboard information is concentrated in the methods **/MakeInterests** and **/KeyEvent**, and the **/KeyMap** dictionary.

/KeyMap is defined using ClassKeyboard to translate the symbolic names **/LeftShift** and **/RightShift** into the keystations specific to the user's hardware. The dictionary maps the key corresponding to **/LeftShift** into the name **/Left**, and similarly for **/Right**.

In **/MakeInterests**, the canvas first creates a FocusSelfInterest, which will trigger whenever this canvas has the focus. (Again, when this interest is expressed, which is done automatically as part of **/activate** to the ClassFrame, it will register the canvas as a focus client.) Then it creates a Keys interest and hangs it off the FocusSelfInterest (using **/addsuite**), so the Keys interest will be active when, and only when, the canvas has the focus.

The Keys interest itself is fairly simple. Since it is not using any shifts as "modifier keys", it uses nulldict as its shiftdict. (Note that this is not the same as using null, which would result in the Keys using a default shiftdict.) The Maps array has only a single map in it, since the shift state will always be zero. The map is the **/KeyMap** dict, which matches events involving the two shift keys.

Thus, while this canvas has the focus, any transitions on the shift keys will match the Keys interest. The Name field in the event will be updated to contain either **/Left** or **/Right**. Then the Action (DownTransition or UpTransition) will cause either the downproc or the upproc to be called. These procs were constructed via buildsend (defined in util.ps); they put a boolean on the stack to indicate the direction of the transition, then call **/KeyEvent** to extract the Name (**/Left** or **/Right**) from the event and store the boolean in the named instance variable.

Complex Example

Finally, here's a particularly complex example. Suppose you don't want the standard keys, but rather you want to see the F3 thru F7 function keys, going up as well as down, and (on them) you have the following requirements for shift-key processing:

- you don't care about the state of Shift or Caps
- you do want to distinguish Meta and Control
- Meta should be treated as a locking shift key

Push it once, it's on; push it again, it's off. You also want these keys to "follow the mouse"; they should be delivered to you whenever the cursor is in your window, regardless of the focus.

This example exercises most of the features of ClassInterest & ClassKeyboard, so let's just work it through all the way, with commentary.

First, assume you've established an event manager as before, either by creating it explicitly or by making your canvas a client of a BaseFrame. The next step is to arrange a Notify interest which is triggered by having the cursor in your window's subtree, independent of the focus:

```
/fni can /new ClassEnterSubtreeInterest send def
```

The keys we're interested in will have different keycodes on different keyboards (or may not exist at all), so we need to ask ClassKeyboard's help in building a basic keymap. The /buildkeydict method was described in an earlier example.

```
/fkeys0 growabledict dup
        {       /FunctionF3 keyforsymbol /FunctionF3
                /FunctionF4 keyforsymbol /FunctionF4
                /FunctionF5 keyforsymbol /FunctionF5
                /FunctionF6 keyforsymbol /FunctionF6
                /FunctionF7 keyforsymbol /FunctionF7
        }       /buildkeydict ClassKeyboard send
def
```

On one keyboard, this defines the following dict:

```
28424:  /FunctionF3
28426:  /FunctionF4
28428:  /FunctionF5
28430:  /FunctionF6
28432:  /FunctionF7
```

This gives us a dict for translating unshifted function keys. Suppose we want Control to override Meta -- if control is down, it doesn't matter whether Meta is on or off. Then we need two more maps, which can be built conveniently from fkeys0:

```
/fkeysC dictbegin
        fkeys0 {
                (Ctrl) exch dup length string cvs append cvn def
        } forall
dictend def

/fkeysM dictbegin
        fkeys0 {
                (Meta) exch dup length string cvs append cvn def
        } forall
dictend def
```

(If we wanted the Control-Meta combination to be distinct, we'd need one more dict; as it is, we'll reuse fkeysC.)

Finally, let's assemble these into a Maps array, for which Meta flips the 1-bit of the shift state, and Control the 2-bit:

```
/fkeymaps [
        fkeys0 fkeysM fkeysC fkeysC
] def
```

That completes the Maps array. But we also need a non-standard ShiftDict, to accommodate our differences from the default, again using **/buildkeydict**.

```
/fshiftdict growabledict dup
        {       /LeftMeta keyforsymbol      [1 /LockChange] buildinterestsend
                /RightMeta keyforsymbol     [1 /LockChange] buildinterestsend
                /Control keyforsymbol       2
        } /buildkeydict ClassKeyboard send
def
```

A reference to the code for ClassShiftInterest may prove helpful here.

The Meta keys tweak the 1-bit, and Control the 2-bit. The Shift interest handles simple on/off shift keys automatically; if the value associated with the key is an int (as is the "2" for **/Control**), the bit gets turned on/off depending on the new state of the key. (If more than one key controls the same bit, e.g., **/LeftShift** and **/RightShift**, then the bit is on if either key is down.) If the value is not an integer, it should be an executable array that computes some off-to-the-side information and then stores 0 in the Name of the event.

LockChange is a utility in ClassShiftInterest, which maintains up to 16 bits of locking-shift-state. It maintains the state on a per-window basis; converting it to have global effect is left, as an exercise, to the reader. (Hint: try demoting Lock-State from an instance- to a class- variable.) The executable matches on **/LeftMeta** and **/RightMeta** are constructed using the buildinterestsend utility in util.ps; the results procs extract the Shift interest from the event and send the **/LockChange** method to it, giving "1" as the lock bit being changed. **/LockChange** also handles storing 0 into the Name field as mentioned above.

Now that we've set things up, we'll actually create some interests using these dicts, and make them available to the event manager.

```
/fki /myFKeyProc dup                % down AND up, remember?
        fkeymaps fshiftdict
           /new ClassKeysInterest send  % create a new KeysInterest
def
fni /addsuite fki send              % get it built into an interest tree
```

We then pass fni back as part of a **/MakeInterests** method, or hand it directly to an event manager (and call addfocusclient) as we did in the first examples.

9. SELECTIONS

9. SELECTIONS

9 Selections

Selections 9-1
Introduction 9-1
- Caveats 9-2
Retrieving Selection Values 9-3
When and How to Transfer a Selection Value 9-8
Making Selections 9-11
Registering a New Selection; Unregistering an Old One 9-17
Responding to Selection Requests 9-18
Utilities 9-20
Class Structure 9-23
Selection Example 9-24

Selections

Introduction

A selection is an indication of some data of interest to the user — almost always, some information visible on the screen which is about to be used in an operation. The most common example is text that is to be moved or copied from one place to another. Many other objects can be selected, and many operations besides move/copy are possible; for instance, a window may be selected so that its properties may be inquired or manipulated. The NeWS Development Environment provides a ClassSelection, whose instances (*Selections*) describe such a selected chunk of data.

The window system has a global registry which keeps track of a few selections; registering a selection causes any previously registered value to be *deselected,* and makes the current selection available to all clients of the window system. This registry is implemented inside ClassSelection, although its facilities are also accessible outside the class system, through utility procedures defined in systemdict. The registered selections are identified by a *Rank,* which may be any non-null PostScript object; the standard ranks are /PrimarySelection, /SecondarySelection, and /ShelfSelection. The ShelfSelection is also commonly referred to as the ClipBoard.

The instance variables for ClassSelection contain attributes of the selection. Some of these are required by ClassSelection's processing: *Holder* is the canvas responsible for the selection, and *Rank* is a global identifier, as described above. Others are attributes which support the user interface for making selections: *Level* is an integer indicating the "size" of the objects selected (for text, 0 - 4 might indicate empty, character, word, line, and paragraph). The full set of UI selection attributes is detailed below. Finally, a Selection usually also contains information stored by the client to identify the selection — e.g., for a text editor, either what the selection contents are, or how to contact the client with a query.

There are two kinds of processing done with respect to selections: Making them and communicating their values. Inquiring the value of an existing selection is relatively easy (and common), so it will be addressed first. This section is primarily concerned with instances of some subclass of ClassSelection.

Making selections is somewhat more complicated, connected as it is to issues of user interface and UI independence. It gets a longer discussion, starting with the section "Making Selections." This section deals both with Selections and with *Selectables* (instances of some subclass of ClassSelectable).

Finally, once a selection has been made, it must expect to have requests posed to it; the last section covers how to respond to such queries.

Caveats

It is important to recognize that a selection can exist without being registered in the global database — instances of (subclasses of) ClassSelection are used privately in several parts of the system before being made available to the world at large.

Another important point is that much of the processing described in this document is initiated outside the application. For instance, global UI code will recognize that a function key has been released, or a Drag action performed, signaling that a selection transfer should take place. Similarly, the UI layer, not the application, is responsible for determining which user actions indicate a selection is to be made or adjusted. This separation is maintained by defining SubClassResponsibility methods in the low-level semantic and UI superclasses, and requiring subclasses which actually get instantiated to implement those methods.

One implication of this second point is that the Selection's methods will often be invoked in some foreign process (the global UI manager, or even in another client's process). They must, consequently, be self-contained — if they need some data such as the connection to the C-side client, that must be reachable from the Selection instance.

One more cautionary note: In order to provide some separation between applications and particular user interfaces (such as the OPEN LOOK user interface), a layer of indirection is inserted into the class structure for Selectables; clients create their own subclass of ClassSelectable by subclassing its defaultclass, not CLassSelectable itself, nor any particular UI's subclass of ClassSelectable. This is explained in more detail below.

Retrieving Selection Values

Applications can retrieve the value of a selection by sending a message to it. This may require that the application first find that selection in the global registry. The relevant methods are

rank	**/getselection**	Selection | null
key	**/query**	false | value true
request-dict	**/request**	response-dict

There is a **/getselection** utility procedure in systemdict, which simply sends the **/getselection** message to ClassSelection. The single argument will normally be one of the Rank names given above, although, as mentioned, it may be any non-null PostScript object. If there is a selection currently registered under that rank, it is returned, else null.

The other two methods above are sent to a Selection; **/request** is the more general (and complex).

A single attribute of a selection can be retrieved most conveniently with the **/query** method. It takes the name of the attribute desired (e.g., /ContentsAscii), and returns the associated value and true, assuming there is such a value; if not, it simple returns false.

/Request is defined to allow multiple requests, and requests with parameters. This may be preferable in several circumstances: when the request is an operation which takes arguments, **/request** must be used. There is also a capability for requester and holder to negotiate the form of the requested data (described below); this also requires use of **/request**. Finally, when the cost of communicating with the holder of the selection is high (e.g. the holder must communicate with its C-side client through a slow communication link in order to respond to any request), it may be advantageous to batch queries in a single call to **/request**.

The argument to **/request** is a dictionary which contains the complete request. Each key in the dictionary names a selection attribute or an operation the selection should perform. For an operation, the corresponding value in the dictionary may be a parameter or array of parameters; for requested attributes, the original value in the dictionary doesn't matter. The selection will return a similar dictionary (or modified copy of the same dictionary, as convenient), with results and requested attributes in the value for each key wherever possible; if it cannot store a result, it will store the value /UnknownRequest.

The following fragments illustrate use of /**query** and /**request**:

```
% to retrieve just the characters of the selection
/PrimarySelection getselection dup null ne {    % sel
        /ContentsAscii /query 3 -1 roll send {          % val
                ...process the value...                         % -
        } if
} {
        pop
} ifelse                                        % -

% a debugger might inquire of an editor where the selection
% is in what source file, so that it could set a breakpoint there
/PrimarySelection getselection dup null ne {    % sel
        dictbegin %  construct request dict
                /FileName null def
                /StartIndex        null def
        dictend                                         % sel request
        /request 3 -1 roll send                         % response
        begin
                ... process response ...
        end
} {                             % no such selection;        % null
        pop
} ifelse
```

The set of request keys passed to the selection holder is open-ended; any set of clients that can agree on the interpretation of a new key, may use that key to communicate among themselves. A convention for the most common requests has not yet been established; but a number of the most useful are suggested here. In general, request names should develop parallel to the conventions of the X11 Window System, as documented in David Rosenthal's *Inter-Client Communication Conventions Manual* (to be distributed by the X Consortium).

Most keys represent requests for the Selection to render its value in a named format. The most common of these is /ContentsAscii; others appear in the table. /ContentsAscii prescribes the selection rendered as a PostScript string, without text attributes (font, typeface, etc.).

Certain keys request that the client modify the selection in some way. Two "operation-type" keys are specified here: /DeleteContents and /ReplaceContents. (Of course, other operations may be defined as clients agree on them.)

/DeleteContents tells the client to delete the contents of the selection. Note that this is not the same as merely deselecting or destroying the Selection instance; e.g., in a text item /DeleteContents means remove the selected span of characters from the text. Since there are no parameters required for this operation, either of **/query** or **/request** will work for it. Assuming the holder is willing and able to comply, a null value will be returned.

/ReplaceContents involves a deletion, just like /DeleteContents; but then new data passed as an argument to the request should be stored in place of the deleted material, and the replacement should be selected. In this case (where the requester must be able to pass an argument to the request), the **/query** method will not work. Instead, the **/request** method is used, with a request dict for its argument. In the request dict, the key /ReplaceContents is defined, with the replacement contents as its value. This style of passing parameters enables a consistent interface to be maintained between requester and selection holder, regardless of the particular requests.

NOTE: No Toolkit selections currently support (or attempt to use) the /ReplaceContents request. It is specified here so that clients who may choose to implement it will have a consistent protocol. The protocol described above matches that in the ICCCM.

Since **/query** retrieves only one selection attribute at a time, the requester can easily control the order in which requests are processed. This is not so easy with **/request**: The order of objects in a dictionary is undefined, so if there is a required order to the requests, the requester must take special pains. It should define only one key in the request dictionary, /RequestSequence, and its value should be an array. The 0th, 2nd, etc. elements of the array will be taken as requests, and the following (odd-numbered) element for each will be the corresponding parameter/value. In the dict returned by **/request**, the value associated with /RequestSequence will be an array in which the odd-numbered elements are the values returned by selection holder.

A similar mechanism allows the requester and holder to negotiate over the form of response. The requester uses the key /RequestChoice, which is defined to an array similar to the one used with /RequestSequence. In this case, the keys in the even-numbered positions of the array are included in the order of the requester's preference. The holder may then choose any of the requests in the

RequestChoice to respond to; the key /RequestChoice is redefined to a new array containing the single key responded to and its corresponding value. If none of the choices is acceptable, the array should be replaced by /UnknownRequest. (If one of the choices in the /RequestChoice array is in turn a /RequestSequence, it is deemed responded to only if all the requests in the sequence are acceptable.)

Like /ReplaceContents, the /RequestChoice key is not currently supported by the NeWS Development Environment. Individual clients may choose to implement it if they are willing to run the risk of having their code become obsolete.

The full details of request processing are described below under "Responding to Requests."

The following table summarizes the conventional request names currently proposed. Those keys marked with * are not currently implemented by The NeWS Development Environment, but are defined so that clients who wish to use such requests will have a common interface. Certain other keys, marked with **, ARE implemented by some or all Toolkit selections, but are retained only for compatibility with the old "Lite" toolkit; their use is not encouraged.

Name	Argument	Result
/Canvas	none	The selected object, if it is a NeWS canvas.
/ContentsAscii	none	A PostScript string containing the selected text, as described above.
/ContentsPostScript	none	A PostScript object, which, when executed, will recreate the selected value. (This is likely to be most useful for graphical objects, which can be redrawn in a new environment.)
/DeleteContents	none	The contents of the selection are deleted, as described above.

Name	Argument	Result
/FirstIndex *	none	The count of how many objects of size *Level* precede the first object in the selection.
/Level *	none	The multi-click level of the selection; see discussion under "Making Selections", below.
/LastIndex *	none	The count of how many objects of size *Level* precede the last such object in the selection.
/RequestChoice *	[request arg ...]	A list of alternative requests (with parameters for each) of which the holder should respond to one, as described above.
/RequestSequence	[request arg ...]	A list of requests (with parameters for each) which the holder should respond to in order, as described above.
/SelectionObjsize **	none	The size of the selection as measured in units of Level 1; e.g., for text, the number of characters. Note the lower-case 's' in Objsize.
/SelectionStartIndex **	none	The count of how many units of Level 1 precede the first such unit in the selection.
/SelectionLastIndex **	none	The count of how many units of Level 1 precede the last such unit in the selection.
/TransferSelection *	dict[... /Source: Selection ...]	One selection is requested to perform a transfer between itself and another; see the note at the end of the next section.

When and How to Transfer a Selection Value

Clients will occasionally decide on their own initiative that they should retrieve a selection value; for instance, the second example above would probably be triggered by invocation of a "Breakpoint" panel button or menu command. But most of the time, global user interface code will determine that a selection transfer is called for. If a client can accept input from the user (keystrokes or mouse drawings, for instance), then it should generally also be ready to accept the contents of a selection. Whenever the UI code determines this is appropriate, it will send an event to the destination of the transfer.

This event's Name is /TransferSelection, and its Action is a dict describing the transfer that is to be made. The most interesting item in this dict will be the key /Source, which will be defined to the Selection whose value should be transferred. The event's Canvas is also significant: if it is null, the event has been "dropped off the cursor," and the value should be inserted as close as possible to the event's location. If the Canvas is non-null, the event was directed to the canvas, not a location; the value should be inserted at the canvas' most recent insertion point. Of course, some canvases may constrain all insertions to a particular location; for instance, a terminal emulator will probably be append-only.

The NeWS Development Environment defines two subclasses of ClassInterest which a client can instantiate in its **/MakeInterests** method, in order to receive and process /TransferSelection events. TransferInterest is an interest in a given canvas being the destination of a /TransferSelection event. The client supplies a method to be called when a transfer event is matched.

```
canvas /methodname /new TransferInterest send        % interest
/methodname                                          % event bool
```

The client's method takes an event and a selection source. It leaves the event unchanged (it is there for examination if needed but should not be consumed). It consumes the selection in sending queries **/requests** to it to effect the transfer, and returns a boolean which is true if the client succeeded in the transfer. If the selection is not of a type acceptable to the client (if, for instance, all queries return /UnknownRequest), the client should return false. This will cause the event to be redistributed further up the canvas tree.

AsciiTransferInterest is a special case of TransferInterest for canvases that require simple text selections. If the selection does not respond to a request for /ContentsAscii, the AsciiTransferInterest refuses to accept the transfer. If /ContentsAscii works, the resulting text is given to the client-supplied method. The client can also specify that the method must be called once per character instead of being called once with the entire string (by passing false as the third argument of **/new**).

```
canvas /methodname stringok? /new AsciiTransferInterest send        % int
/methodname                                                          % event string
```

Thus, most clients should never have to worry about constructing an interest in /TransferSelection, nor about terminating the transaction in which the selection is transferred. Typically it suffices to put a single line in your canvas's **/MakeInterests** method, such as:

```
Canvas /inserttext true     /new AsciiTransferInterest send
```

and then write a suitable **/inserttext** method (in this example, one that consumes a string).

Clients which construct their own interest in /TransferSelection events must satisfy a few requirements:

1. When they receive such an event, they should make appropriate requests to the Source selection, just like clients using a TransferInterest.
2. If they cannot accept the selection, they must redistribute the /TransferSelection event themselves, to give ancestor canvases a chance to deal with it.
3. If the transfer succeeds, the recipient should complete the transaction by sending a message to the UI code:

```
description-dict /transferfinished /defaultclass ClassUI send send
```

or

```
description-dict /transferfinished MySelectable send
```

where MySelectable is any subclass of [/defaultclass ClassSelectable send]. The **/transferfinished** method requires the description-dict from the /TransferSelection event's Action on the stack (and consumes it). This method is needed because only the recipient knows when it is finished with the transfer, and various cleanup actions (such as deleting the Source if the transfer is a Move vs a Copy, or unhighlighting the source, etc.) must wait until the transfer is over.

Implementation note: It is possible for a Selection to be sent a /TransferSelection *request* directly, rather than through an event. That is, a selection may receive a request whose key is /TransferSelection; the associated value will be a description-dict just like the one contained in the Action of a /TransferSelection event. In this case the Selection receiving the request is expected to insert the value at its current insertion point — there is no convention for passing specific coordinates. Since there is no UI code driving the transfer, the destination Selection must attend to other details: it should delete its own current value if its /PendingDelete? attribute is true; it should send a /DeleteContents request to the Source Selection if the key /DeleteSource? in the transfer description-dict is true; if either it or the Source has Rank of /SecondarySelection, it should send **/deselect** to that. This latter form of transfer is intended to handle operations such as exchanging two selections. However, no such requests are currently being generated by the Open Look UI manager, and no client Selections currently support such transfers.

Making Selections

A selection client is a canvas which can contain a selection, or be one itself (e.g. a text window or a frame). The NeWS Development Environment provides utility classes to cover common text selection clients (DragTextSelectable) and also for the frame and icon windows for applications, which are selectable objects. Application programmers will want to avail themselves of these if possible; they are described at the end of this document.

Other selection clients will subclass 2 classes: ClassSelection, introduced above, and (a subclass of) ClassSelectable, whose instances are interests in user actions. In both, there are SubClassResponsibility methods which the client must implement in its own subclass.

This discussion of making selections will probably be more intelligible after a high-level sketch of the protocol. Here is a typical sequence of operations for a selection-client application:

- The application creates a new Selectable. This is usually done inside a canvas's /MakeInterests method, so that the general canvas mechanism handles forking an event manager and expressing the interest (the Selectable).
- UI-specific code inherited in the Selectable matches certain events (which events get matched depends on the particular UI), and decides a selection action has occurred.
- The UI-specific code calls a SubClassResponsibility method (**/newselection**); the client's Selectable subclass returns a new Selection of the appropriate class (i.e., the client's Selection subclass).
- The UI code then sets instance variables within that Selection to indicate what's going on (multiclick level, etc.).
- Next it calls more SubClassResponsibility methods to get the client to finish resolving the operation. E.g., the client might need to resolve mouse coordinates into a character index and then highlight the selected text.
- When the UI code decides that the user action is complete (e.g., a mouse button is released), it registers the Selection in the global dictionary so that other applications can access it.

Note that most of the goings-on are driven by the UI code. Clients need write relatively little code for their particular selections. In particular, clients should *not* try to interpret raw device events - some are not readily accessible to the client (/Copy key when the client holds the selection, but not the focus); most are subject to user-modification (some left-handers swap the meaning of the mouse buttons and function keys for left-handed use); and user interfaces are subject to many changes an application will do well to ignore if it can.

Selections have been introduced already. There is more to be said about them, but first we consider Selectables.

A Selectable is a NotifyInterest on a single canvas; it watches for some event in a small set which initiate making a selection (e.g., button-down on the Point or Adjust buttons). When an initiating event is seen, it expresses interests in other events like mouse motion and the corresponding button-up, and starts up some state machinery which implements the current user-interface for making selections. It also causes creation of a new Selection, to be used in communicating to the client about the selection that is being made; see the **/newselection** method.

The definition of a subclass of ClassSelectable should be parameterized as follows, to avoid wiring a particular User Interface into the application.

```
/MySelectable [/defaultclass ClassSelectable send]
        [<instance vars>]
classbegin
```

The SubClassResponsibility methods in a Selectable are:

event rank holder	**/newselection**	selection
event selection	**/selectat**	–
event selection	**/adjustto**	–
event selection	**/dragat**	–
event selection	**/dragto**	–
event selection	**/inselection?**	bool
event selection pos	**/attachinsertionpoint**	–

Again, most clients will also elaborate

holder canvas type	**/newinit**	interest
–	**/destroy**	–

The client makes a Selectable instance for each selectable canvas, typically by sending **/new** to a subclass of Selectable as part of the canvas's **/MakeInterests** method. The 'type' parameter tells the global UI code what sort of UI to use for selections on this canvas. There are currently three types defined; others may be added as needed. (The UI manager will invoke some reasonable default behavior for any unrecognized type.) The known types are:

/Text	Text within the canvas can be selected
/Graphics	Graphics objects within the canvas can be selected
/Canvas	The canvas itself can be selected (e.g., a frame)

The holder should be a canvas which uniquely identifies this client; it will be the same as the Holder attribute of selections made in this canvas. It is used by ClassSelectable in an identity test, to determine whether an existing selection is held by this client. Normally, the holder and canvas arguments to **/new** will be the same. But if a single selection may exist in two or more canvases (e.g. a split view in a text editor, or several icons on the desktop), then the Selectables for those two canvases should have the same holder. This supports such behavior as starting a selection in one canvas of a split view and then extending it by clicking in the other canvas.

When a Selectable interest is satisfied, its activation procedures will often start a new selection. This involves a send to the client's **/newselection** method, with appropriate rank and holder arguments. This method in the Selectable should create a new Selection instance (by sending **/new** to the client's subclass of ClassSelection). The returned instance gets its attributes filled in by the UI layer of Selectable, and is then passed back to the client in sends to its other SubClassResponsibility methods.

The other six SubClassResponsibility methods of a Selectable all take an event and a Selection instance. (One method also takes a third parameter.) The event is useful only for its coordinates – extracting the Coordinates array from an event automatically transforms them according to the current canvas and graphics context. The Selection contains all the other parameters of the method, as described later. The methods should operate as follows:

`/selectat`

`event selection` **/selectat** –

Resolve the coordinates of the event to an object, and start a selection on it with the given attributes. Note again that the Selection instance will already have been created via a separate call to **/newselection.**

`/adjustto`

`event selection` **/adjustto** –

Adjust the boundary of the given selection to lie on the object at the event's coordinates, again attending to the attributes in the selection.

`/dragat`

`event selection` **/dragat** –

Initiate user feedback for a Drag (direct-manipulation move or copy) of the given selection; e.g., start an overlaid image of the value being dragged). If a grasp-point is needed (i.e., if the cursor coordinates are needed in order to position the feedback), use the coordinates of the given event.

`/dragto`

`event selection` **/dragto** –

Move a Drag-image so its grasp-point is at the coordinates of the given event, or give other feedback of a Drag in progress.

`/inselection?`

`event selection` **/inselection?** `bool`

Return true or false as the location of the given event is, or is not, within the given selection. (The global UI code uses this to resolve multi-clicks or other special behavior resulting from clicking within an existing selection.)

`/attachinsertionpoint`

`event selection`
`postionname` **/attachinsertionpoint** –

Note the specified point for later use as an insertion point if a value is copied to the selection. In particular, if this is a primary selection, the canvas should usually change its input focus

location. The positionname, together with the position of the event and the endpoints of the given Selection, are interpreted as for the /ComputeNamedPosition utility, described later. Unlike the five other methods listed above, this one actually has a default implementation that simply ignores (pops) its arguments.

Now we return to ClassSelection, for the attributes used in the selection-making process. These are:

Key (Name)	Value Type	Interpretation
/Level	int	The "size" of objects to be selected. For instance, in text, 1 may indicate a character, 2 a word, 3 a line or sentence, etc. Essentially, for OPEN LOOK, Level is a multi-click count.
/PendingDelete?	bool	True if the selection should be replaced by the next user input action (always, for primary selections in OPEN LOOK).
/DeleteSource?	bool	Meaningful only if the selection is being passed to a `/dragat` or `/dragto` method, in which case the key is true if the operation is a Move rather than a Copy.
/Pin	any	Tells the client where to "anchor" the selection during an `/adjustto` operation. The pin can be a name, in which case it can be evaluated using /ComputeNamedPosition (described below under "Utilities"), or it may be an arbitrary value (typically an int) representing a previously computed anchor point.

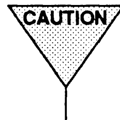
Key (Name)	Value Type	Interpretation
/Preview?	bool	True if the selection is still being adjusted by the user; when the user finishes (e.g., releases the mouse button), the client will get a last `/selectat` or `/adjustto` message with /Preview? set to false. Many clients will ignore /Preview?; it is provided as an accelerator for clients that wish to postpone moving some selection info to a more permanent location until the selection settles down.
/Rank	any	Most selections have Rank eq /PrimarySelection. Selections made while some function-keys are down have Rank eq /SecondarySelection. (These get reflected differently, and have special uses.) The Clipboard has Rank eq /ShelfSelection. Other Ranks are possible, though not currently used.
/Registered?	bool	True if the selection has been registered in the global database via `/setselection`.
/Hilited?	bool	This key is defined only if the Type of the Selectable which made the selection is /Canvas. For these, it shows the previewed-state (as distinguished from the true state) of the selection. E.g., Adjust-down on an icon [de]-hilites it immediately, and then toggles its hiliting as you slide off & on the icon.
/Style	any	The style of highlighting recommended by the UI manager. Currently defined values are /Default, /Invert, /Outline, /StrikeThru, and /Underscore. Clients may ignore this value if they think it does not apply to their selection type, basing their highlighting instead on /Rank and /PendingDelete?.

Finally, to assist clients in correctly reflecting changes to the selection, every time the /adjustto message is sent, the accompanying selection will have the name /Changed defined

to an array of some of the above names; each key in the array has changed value since the previous adjustment.

Notes:

It is possible for an **/adjustto** to indicate a point outside the area in which contents can be displayed (e.g. off the bottom of a text window). This supports an auto-scroll feature, such as defined by the OPEN LOOK user interface. When an application gets such an **/adjustto**, it should (if possible) scroll some new data into the visible region from the hidden region indicated by the location of the **/adjustto**, and select everything up to that border. It should repeat this process as long as **/adjustto** messages continue to be received.



autoscroll doesn't work yet — new messages are not generated. Clients willing to be obsoleted in the next release can repeat the scroll operation on their own until an **/adjustto** is received with coordinates inside the canvas, or with /Preview? set to false.

Registering a New Selection; Unregistering an Old One

When the UI layer decides that a selection specification has been completed, it sends **/setselection** to the Selection; the default implementation registers the instance in the global database. Clients may override **/setselection** if they need to adjust state or maintain any additional information when one of their selections becomes publicly available. Of course, it is also possible for a client to create a new selection on its own initiative and send it **/setselection**; it will get registered just the same.

When a new selection is registered, any old selection already registered under that rank is sent a **/deselect** message. **/deselect** is a strict SubClassResponsibility method: there is no implementation in ClassSelection. The subclasser's method should at least de-hilite the selection; most selections will also destroy themselves. However, a deselected Selection might be retained, for instance to support restoring the selection in an Undo operation.

Just as a client may register a selection on its own initiative, it can unregister one by sending **/unsetselection** to it. A warning given above bears repeating here: It often happens that a **/deselect** message is generated in the client which is making a new selection (the cause of this one being deselected); the message-send in this context is in danger of communicating with the wrong client. For instance, printing data over currentprocess' stdout is likely to send it over the wrong connection. Clients must take care to store some access back to their generating client in selections they create, so they can communicate with it reliably when that selection receives a message while running in another process. Other messages that may be sent to a selection face analogous dangers: **/singlerequest** and **/destroy** in particular are liable to be sent to a selection while running in some process other than the selections' Holder's client. Equivalent care is required in these cases to respond in the proper context.

Responding to Selection Requests

Selections respond to requests through either of two methods:

```
       request-dict   /request         => response-dict
oldval request-key    /singlerequest   => newval
```

Subclassers must override AT LEAST ONE of these two methods.

The first has been described from the requester's point of view above. The default implementation in ClassSelection is in terms of **/singlerequest**: the request-dict is enumerated with forall, and each request/value pair is passed to **/singlerequest.** (If /RequestSequence is found, it is passed to its own enumerator, which in turn calls **/singlerequest**. When /RequestChoice is implemented it will work similarly.) If a selection holder wishes to batch processing of requests, it should override the **/request** method. Any such override is then responsible for supporting /RequestSequence and /RequestChoice.

Instead of overriding **/request**, a subclasser may choose to override **/singlerequest**. The key passed to a Selection's **/singlerequest** method identifies the nature of the request. Most keys represent requests for the Selection to render its value in a named format. For these requests, the value currently on the stack is to be discarded, and the client should put the requested value on the stack.

Certain keys request that the client modify the selection in some way. For these requests, the oldval on the stack contains arguments, if any, to be used in the operation. Even if no arguments are needed, the **/singlerequest** method must be sure to remove the value from the stack; likewise, even if there is no return value, **/singlerequest** must store something (typically 'null') on the stack. This ensures a uniform interface to **/singlerequest**. (If neither the oldval nor the newval is meaningful, the oldval can simply be left on the stack as the returned value. See the /DeleteContents case in the example below.)

For any request (value or action), the client may choose not to support the requested key. If so, **/singlerequest** should pop the oldval and return /UnknownRequest.

Most clients will choose to override **/singlerequest** rather than **/request**, since the code is considerably simpler. A typical **/singlerequest** method might look like:

```
/singlerequest { % oldval key => newval
{       /ContentsAscii {pop Rank /GetValue Holder send}
        /SelectionObjsize {pop Rank /GetValue Holder send length}
        /Level {pop Level}
        /Canvas {pop Holder}
        /DeleteContents {Rank /DeleteSel Holder send} % leave junk val
        /Default {pop /UnknownRequest}
} case
} def
```

Note that **/singlerequest** is not required to support /RequestSequence or /RequestChoice. Since clients making such requests must always go through the **/request** method (rather than **/query**), it is left to the **/request** method to handle breaking up the /RequestSequence or /RequestChoice into a series of calls to **/singlerequest**.

Utilities

The NeWS Development Environment defines five subclasses of particular interest to clients and/or implementors of selections. Two of these, TransferInterest and AsciiTransferInterest, have been described above. The remaining three comprise two convenience subclasses of ClassSelection, and a subclass of ClassSelectable which provides some of the basic functionality of selectable text.

StaticSelection is a selection whose value never changes; it responds to queries by looking up the queried keys in a constant dict. When a selection is copied to the Clipboard, the UI manager sends **/checkpoint** to the Selection to obtain a static copy. The default **/checkpoint** method calls **/Checkpoint**, which in turn sends a **/request** to obtain the value of the selection in several formats, and stashes the results in a StaticSelection. Clients may override **/checkpoint** if they are willing to take responsibility for maintaining a static copy of their value.

E.g., a text window might prefer not to copy a potentially large string to the Clipboard, but could instead cache the necessary pointers back to its data. However, if the text window's data subsequently changes, the text window must be sure that the static copy is not affected. If the static copy is about to become invalid, the text window can call **/Checkpoint** (mixed case) to fall back on creating a StaticSelection.

StringSelection is a special case of StaticSelection that only knows how to render itself as ContentsAscii. This is to make it easy for clients to wrap a string inside a selection preparatory to handing it to a canvas via a /TransferSelection request. (See **/sendtocanvas**, below.) A StringSelection is intended to be created directly via **/new** instead of via a ClassSelectable, and thus does not expect rank/holder arguments; its **/new** method takes just a string.

DragTextSelectable provides assistance for clients whose selections are character strings, and who want to use an overlay canvas to display the selection during a drag-move or drag-copy operation. It fills in the **/dragat** and **/dragto** methods for you. Subclassers will generally wish to override the **/CurrentText** method for greater efficiency (the default uses the normal **/query** mechanism, whereas individual subclasses can usually obtain the text by more direct methods). Subclassers may also need to override /CurrentFont, if the TextFont of their Selection's Holder-canvas is not suitable.

Assorted utility methods:

/ComputeNamedPosition { % first last current positionname => pos

In ClassSelectable. First, last, current, and pos are numeric values referring to the location of a selection (in the client's interpretation); first/last are the endpoints of an existing selection, while current is typically the position corresponding to the coordinates of a recent event. Positionname is one of the values defined for the /Pin in a Selection, and is interpreted as follows:

/LowEnd	the low end of the existing selection (first)
/HighEnd	the high end of the existing selection (last)
/NearEnd	whichever of first/last is closer to current
/FarEnd	whichever of first/last is further from current
/AtPoint	the cursor position (current)

This is used for interpreting **/attachinsertionpoint**, and also for interpreting the /Pin to establish one endpoint in preparation for subsequent **/adjustto** messages.

/computepin { % first last current => pin-point

In ClassSelection. If the Selection's /Pin is a name, **/computepin** calls **/ComputeNamedPosition** and stores the result as the new value of /Pin. Otherwise the previously computed /Pin value is returned unchanged. This should be done on every **/selectat** or **/adjustto**; the UI manager will override /Pin again if the anchor is to change.

/computerange { % first last current => newfirst newlast

In ClassSelection. Same as **/computepin**, but it returns the pinned position and the current position, in sorted order.

/checkpoint { % rank => selection

/Checkpoint { % rank => selection

In ClassSelection. See earlier discussion re StaticSelection.

/CanRenderAs { % - => namearray

In ClassSelection. A list of names describing the formats in which the selection might be able to render itself. The list is used by /Checkpoint to decide the formats to install in the StaticSelection. It's okay for the list to include formats that the Selection in fact cannot handle; they'll just

get mapped to /UnknownRequest in the static copy. The default list is:

/ContentsAscii	/SelectionObjsize
/ContentsPostScript	/SelectionStartIndex
/SelectionLastIndex	

Subclassers can override this to extend (or truncate) the list, so that stuffing their selections to the Clipboard will include all appropriate formats. NOTE: The list should NOT include /Canvas, even though the selection's **/singlerequest** method might handle such a request. This is because copying a /Canvas value to the ClipBoard can result in the canvas staying on the screen after its application has been destroyed.

/transferfinished { % tdict => -

In [/defaultclass ClassUI send] (or any descendant thereof). Handles all cleanup following a transfer, such as deleting the source if the transfer was a Move, unhighlighting the source if appropriate, etc.

Called automatically by TransferInterest and AsciiTransferInterest, so most clients need not worry about it.

/sendtocanvas { % canvas [delete?] => -

In ClassSelection. Sends the selection to the given canvas via a /TransferSelection event. The selection need not be registered with the global manager. The optional bool says whether the selection should be deleted after the transfer. (Default is false.) The canvas can be null to send the event to the canvas(es) currently under the pointer. For example, the following would send a string to the current input focus:

```
(random string) /new StringSelection send
currentinputfocus /sendtocanvas 3 -1 roll send
```

```
/query {    % key    => value true
            %        => false
```

In ClassSelection. See discussion under Transferring & Querying.

/getselection { % rank => sel | null

In systemdict. Looks up the given rank in the global registry and returns the Selection (if any) currently registered for that rank.

E.g.,

```
/PrimarySelection getselection
```

/clearselection { % rank => --

In systemdict. Removes from the global registry the Selection (if any) currently registered for the given rank.

Class Structure

This section should be of interest only to UI implementors, i.e., people who want to supplant the Open Look UI manager with a different look and feel. All other readers may skip this part.

Implementing a specific look and feel involves subclassing three classes: ClassUI, ClassSelectable, and ClassFunctionKey. This is the only time that those classes should be subclassed directly; normal clients should always subclass from [/defaultclass ClassXXXX send]. A look and feel implementor will subclass ClassUI, then use that subclass as a mix-in when creating the other two subclasses. For example, ClassOLSelectable inherits from ClassSelectable and ClassOLUI.

Having defined the three subclasses, you can install the new look and feel by sending /InstallUI to the subclass of ClassUI, e.g., /InstallUI ClassOLUI send. This builds three new subclasses: ClassSpecificUI is a subclass of the given subclass of ClassUI. ClassSelectableUI inherits from the subclass of ClassSelectable, and similarly for ClassFunctionKeyUI. These three new subclasses are actually

the /defaultclasses for ClassUI, ClassSelectable, and ClassFunctionKey; thus they are the classes that all clients subclass from.

(The function key portion of this hierarchy is beyond the scope of this document.)

The purpose of the additional layer of subclassing is to enable a new look and feel to be installed in a running system. Redefining an existing class reuses the class dictionary, so every ClientSelectable's superclass pointer to ClassSelectableUI will always remain valid.

Selection Example

What follows is a lengthly example demonstrating how selections work in the NeWS Development Environment. Also included on the tape that this document was on is this selections example as an executable. On the tape the name of this example is selections-example.ps.

Example

```
% This is a complete application that demonstrates much of what's involved in
% supporting making/manipulating selections within a canvas.  The application
% brings up two instances of a window containing a few circular subcanvases
% (which scale automatically with the parent window).
%
% You can select circles by dragging out a bounding box that encloses the
% CENTER of the desired circle.  (It wouldn't be hard to add selection of
% single circles by clicking on them, but it wouldn't be any more
% instructive.)  If you drag out a box using the "adjust" mouse button, it
% adds/removes the enclosed circles from an existing selection.
%
% Per OPEN LOOK, if you mouse down inside an existing selection and move the
% mouse, it drags the selection to a new location.  (Holding down the CONTROL
% key while you do this causes the selection to be copied instead of moved.)
% You can drag between two different Circles windows; you can also use cut and
% paste.  Secondary selections (quick-copy/move) are NOT supported for this
% example.
%
% Comments throughout the code explain how each piece fits into the
% selection paradigm.
```

(continued on next page)

```
% First, the parent canvas.  It is a Bag that will contain the circular
% canvases as children.  The "baggage" for each child is the x/y of its
% center, and the radius.  The parent uses a 1x1 coordinate space, so
% the children automatically rescale with the parent.  The first several
% methods -- down through /Layout -- are stuff for the bag and have
% nothing to do with selections.

/CirclesCanvas ClassBag nullarray
classbegin

     /FillColor .5 dup dup rgbcolor def  % 50% gray
     /BorderStroke 0 def                 % default (2) is bogus in our coordinate space
     /Transparent false def              % don't flash background when removing circles
     /Mapped true def                    % override default (false for opaque canvases)

        % Override: create some initial circles.  The "gray" parameter is
        % consumed by /new to CircleChild; the x/y/radius is our "baggage".
        %
        /newinit {
                %      X  Y radius gray
                null [.3 .3   .1    .35 CircleChild] /addclient self send
                null [.8 .5   .15   .7  CircleChild] /addclient self send
                null [.5 .9   .05   .85  CircleChild] /addclient self send
        } def

        % Override: Cause canvas to be some reasonable size by default.  Note
        % that /minsize is not overridden, so the canvas CAN be made smaller.
        %
        /preferredsize { % - => w h
                300 300
        } def

        % Override: Give this canvas a 0-1 0-1 coordinate system.
        %
        /Transform { % x y w h => x' y' w' h
                4 2 roll        % w h x y
                translate       % w h
                scale           % -
                0 0 1 1
        } def

        % Override: Lay out our children according to the baggage.  Since the
        % children's /reshape method wants x y w h, we have to convert the
        % center coords and radius into a rect, even though the child will
        % immediately convert them back.  Such is object-oriented life.
        %
```

(continued on next page)

```
/Layout {
	/clientlist self send {
		dup /baggage self send		% can [cx cy r]
		aload pop 3 1 roll		% can r cx cy
		2 index sub exch 2 index sub exch	% can r x y
		3 -1 roll 2 mul dup		% can x y w h
		/reshape 6 -1 roll send
	} forall
} def

% Override: Add selection-related interests. CircleSelectable is the
% interest in selection-causing events. TransferInterest is the
% interest in being on the receiving end of a drag or paste. The
% ClassFocusSelfInterest is necessary so that we can become the input
% focus, because the "paste" key sends a /TransferSelection event to
% the focus, not to the window the mouse is over. That is, the
% TransferInterest will see a /TransferSelection event if it gets
% sent to us, but we have to be the focus in order for the event to
% get sent in the first place.
%
% If the ClassFocusSelfInterest were removed, the ONLY effects would be
% (a) the circles window would never take over the input focus, and
% (b) you couldn't paste circles into a circles window after deleting them.
%
/MakeInterests {
	/MakeInterests super send
	self dup /Graphics /new CircleSelectable send
	self /ReceiveTransfer /new TransferInterest send
	self /new ClassFocusSelfInterest send
} def

% Toggle the selectedness of each child contained by the given rect.
% This method is invoked from CircleSelectable. (It could have been
% implemented directly in CircleSelectable, but since it needs to
% invoke several methods of CirclesCanvas it is more efficient to
% package them up as a single "send"; i.e., it's much faster to do
% a single send that then does several "self send"s.
%
% Note that ToggleSelection in turn sends to the individual children to
% change their selectedness; many selection clients, of course, do not
% have subcanvases and maintain all their selection state within a
% single instance. See ClassTextControl for an example.
%
/ToggleSelection { % x y w h => -
	/children+ self send {
		5 copy ChildInRect? {		% x y w h child
```

(continued on next page)

```
                        /ToggleState exch send
                ) {
                        pop
                ) ifelse                        % x y w h
        ) forall
        pop pop pop pop
) def

% Return true if child is contained by given rect. To keep the example
% simple, we return true if the child's center is within the rect,
% rather than testing for complete enclosure. Of course, this makes
% it impossible to select just one of several concentric circles.
% But hey, it's just a demo!
%
/ChildInRect? { % x y w h child => bool
        /baggage self send      % x y w h [cx cy cr]
        aload pop pop           % x y w h cx cy
        0.0001 dup rectsoverlap
} def

% Receive a selection via a /TransferSelection event. TransferInterest
% (see /MakeInterests) will call this method when such an event is seen.
% We query the selection to see if it's a "Circles" selection; if not,
% we return false and TransferInterest redistributes the event for us.
% (E.g., if an icon is dragged over a Circles window and released, it
% will fall through to the desktop.) Assuming the selection IS Circles,
% we add the circles to our canvas, translating them according to the
% location of the event. (We apply min/max to keep them from falling
% outside the canvas, lest they be lost forever.)
%

% NOTA BENE: The newly inserted circles do NOT end up being selected.
% If the operation was a Move, nothing ends up selected; if it was a
% Copy, the source stays selected. Clients might well wish to select
% the newly inserted stuff, but if the /ReceiveTransfer method were to
% do this it could cause trouble if the operation is a Move, because the
% global mechanism will subsequently try to delete the source Selection,
% which might no longer exist! This is a recognised design deficiency
% that will be addressed after the initial release.
%
   /ReceiveTransfer { % event sel => event bool
       /Circles /query 3 -1 roll send {        % event [circles]
           gsave
                Canvas setcanvas
                1 index /Coordinates get aload pop    % ev [circs] dx dy
                3 -1 roll {                           % dx dy [x y r gray]
```

(continued on next page)

```
                                aload pop 4 -1 roll            % dx dy y r gray x
                                5 index add 0.005 max 0.995 min  % dx dy y r gray x'
                                4 -1 roll                      % dx dy r gray x' y
                                4 index add 0.005 max 0.995 min  % dx dy r gray x' y'
                                4 2 roll                       % dx dy x' y' r gray
                                4 copy pop 0 360 arc extenddamage
                                CircleChild 5 array astore
                                null exch /addclient self send  % dx dy
                            } forall
                            pop pop true
                    } {         % event
                            false
                    } ifelse
            } def

classend def

% Next we have the circular children themselves.  They use /BorderStroke
% to note whether they're selected, by promoting a non-zero value.  (Note
% that the value used is very small, because it's interpreted on the parent's
% 1x1 coordinate system.)  A few simple methods are provided for toggling
% or clearing the selected state.  The rest of the class has to do just
% with implementing the circular border.

/CircleChild ClassCanvas nullarray
classbegin

        /BorderStroke 0 def        % promoted to 0.01 if selected

        /newinit {                 % gray => -
                /FillColor exch dup dup rgbcolor promote
        } def

        /Selected? { % - => bool
                BorderStroke 0 ne
        } def

        /ToggleState { % - => -
                /BorderStroke Selected? {unpromote} {.01 promote} ifelse
                /paint self send
        } def

        /KillSelection { % - => -
                Selected? {ToggleState} if
        } def
```

(continued on next page)

```
        % Override: Circular border.  Note that incoming x/y are lower left, not
        % center.
        %
        /path { % x y w h => -
                min 2 div 3 1 roll                      % r x y
                2 index add exch 2 index add exch       % r cx cy
                3 -1 roll 0 360 arc
        } def

        % Override.  Default method sometimes suffers from roundoff errors.
        %
        /StrokeCanvas { % color inset => -
                2 {neg dup /bbox self send insetrect /path self send} repeat
                pop setcolor eofill
        } def

classend def

% This is the guts of the selection stuff for this demo.  For the most part
% it just implements the required "SubClassResponsibility" methods.  There
% are a couple utility methods at the end.

/CircleSelectable [/defaultclass ClassSelectable send] nullarray
classbegin

% Class Variables:

        % These three values are overridden when an operation is in progress
        % that involves an overlay canvas for feedback.  The two operations
        % that do so are (1) bounding box for making/adjusting a selection,
        % and (2) dragging an existing selection to a new location.

        /OverlayCanvas null def         % promoted when in use (bounding box and drags)
        /X null def                     % initial X of bounding box (promoted)
        /Y null def                     % initial Y of bounding box (promoted)

% Methods:

        % Subclass responsibility: Create an instance of our subclass of
        % ClassSelection.  No special initialisation is required.
        %
        % Note that, as with the other subclass responsibility methods, this
        % demo code does not itself contain any sends to the /newselection
        % method.  CircleSelectable INHERITS code that, having decided that
        % a particular user action should be interpreted as starting a new
        % selection, then does "/newselection self send", thereby calling the
```

(continued on next page)

```
% method below.
%
/newselection { % event rank holder => selection
        /new CircleSelection send
        exch pop
} def

% Subclass responsibility: Update a CircleSelection instance to reflect
% the start of a new selection.  For our purposes this is treated the
% same as when we start adjusting an old selection, so we share the
% code with that method.
%
/selectat { % event selection => -
        /adjustto self send
} def

% Subclass responsibility: Update a CircleSelection instance to reflect
% a new selection action.  Many of the variables in the Selection carry
% information passed to us from the global UI code.  In particular, if
% /Preview? is true, then the user is still marking the selection, and
% we don't need to change anything except the bounding-box feedback.
% (If we wanted to get fancy, we could adjust the highlighting of the
% circles on the screen but not actually change whether they're selected
% yet, but let's not be fancy.)  If /Preview? is false, the operation is
% over and it's time to change the state of the affected canvases.
%
% Note that we support only PrimarySelection; if we're asked to make a
% SecondarySelection we ignore the request and leave the selection empty.
%
% As mentioned earlier, when the operation is finished (Preview=false),
% we use a method in CirclesCanvas to do the actual toggling, to avoid
% doing too many cross-instance sends.
%
/adjustto { % event selection => -
        /Rank 1 index send /PrimarySelection ne {
                pop pop
        } {
                OverlayCanvas null eq {
                1 index Canvas StartOverlay
                } if
                exch EventCoords X Y points2rect            % sel x y w h
                /Preview? 6 -1 roll send {                  % x y w h
                        gsave
                                OverlayCanvas setcanvas erasepage
                                0 setgray 0 setlinewidth rectpath stroke
                        grestore
```

(continued on next page)

```
                } {                                     % x y w h
                        EndOverlay
                        /ToggleSelection Canvas send
                } ifelse
        } ifelse
} def

% Subclass responsibility: When a drag operation begins we just remember
% the initial coords and create the overlay. Note the overlay covers the
% whole framebuffer (unlike in /adjustto), but the coords are in our own
% CTM. We use a temp event's Coords in /dragto to convert between them.
%
/dragat { % event selection => -
        pop framebuffer StartOverlay
} def

% Subclass responsibility: Give feedback for the drag in progress.
% We won't try to do anything fancy like showing the actual circles
% (hard to decide how to scale them anyway when moving between windows);
% we just draw a line from where the drag started to the current point.
% When /Preview? is false, the operation is over, so we tell the
% selection to remember the offset where the drag began (we could've
% done that in /dragat instead).
%
% Note that we don't actually perform the transfer operation here;
% /dragat and /dragto are mainly involved in feedback to the user.
% The /TransferSelection event is generated by the global UI machinery.
%
/dragto { % event selection => -
    /Preview? 1 index send {
        pop
        gsave
            % Use a temporary event to convert X/Y from the original
            % canvas's CTM to that of the overlay.
            Canvas setcanvas
            createevent dup /Coordinates [X Y] put        % ev evtemp
            OverlayCanvas setcanvas                       % ev evtemp
            /Coordinates get aload pop                    % ev x' y'
            erasepage 0 setgray 0 setlinewidth moveto     % ev
            /Coordinates get aload pop lineto stroke
        grestore
    } {
        X Y /SetDeltas 4 -1 roll send                     % ev
        pop
        EndOverlay
    } ifelse
```

(continued on next page)

```
} def

% Subclass responsibility: Normally we'd extract the coordinates from
% the given event and see if they're inside the given selection.  But
% in this demo we have the advantage of knowing that the selected
% objects are canvases, so we can let NeWS do a lot of the work for
% us by using canvasesunderpoint.  As before, note that we uniformly
% return "false" if the selection is not Primary.
%
/inselection? { % event selection => -
     /Rank exch send exch pop false exch              % false rank
     /PrimarySelection eq {
          null canvasesunderpoint {                   % false can
               % Look for canvas whose parent is the CirclesCanvas.
               dup /Parent get Canvas eq {
                    % Found it: Is it selected?  In any case, exit the loop.
                    /Selected? exch send or
                    exit
               } {
                    pop
               } ifelse                               % in?
          } forall
     } if
} def

% Given an event, obtain its X/Y in our canvas's coordinate space.
%
/EventCoords { % event => x y
        gsave
                Canvas setcanvas
                /Coordinates get aload pop
        grestore
} def

% Start an operation that uses an overlay canvas.  The overlay sits on
% either our own canvas or the whole framebuffer.  This method also
% obtains the coordinates from the initiating event (using our canvas
% for the CTM) and stashes them for later use.
%
/StartOverlay { % event canvas => -
        createoverlay /OverlayCanvas exch promote          % event
        EventCoords /Y exch promote /X exch promote
} def

% Clean up after an overlay operation.
%
```

(continued on next page)

```
        /EndOverlay { % - => -
                gsave OverlayCanvas setcanvas erasepage grestore
                /OverlayCanvas unpromote /X unpromote /Y unpromote
        } def

classend def

% The ClassSelection subclass again implements the required methods.
% Commonly these are implemented by short bits of code that simply
% call similar methods in the Canvas (which is available as the /Holder
% instance variable in most Selections). Here, though, the operations
% are simple enough that most of the work is done directly.

/CircleSelection ClassSelection nullarray
classbegin

        % Override: This class variable tells the global machinery what forms
        % of query to stash away when copying one of our selections to the
        % clipboard (i.e., for cut & paste).
        %
        /CanRenderAs [/Circles] def

        % The following methods are promoted to constants when a drag operation
        % is performed. The defaults normalise all the selected circles to be
        % at 0,0 (which is then offset by the location of the "get" action).
        % When a drag is performed, the promoted constants cause the destination
        % coords to be interpreted relative to the starting point of the drag.
        %
        /DeltaX {dup} def       % offset due to drag (promoted)
         /DeltaY {dup} def      % ditto

        % Subclass responsibility: Remove all circles from this selection,
        % thereby dehighlighting them. We do this by asking our Holder (the
        % CirclesCanvas) for a list of its children, and then telling each
        % child to deselect itself.
        %
        /deselect { % - => -
                Rank /PrimarySelection eq {
                        /children+ Holder send {
                                /KillSelection exch send
                        } forall
                } if
                /DeltaX unpromote /DeltaY unpromote
        } def

        % Get a list of the CircleChild canvases that are selected in our
```

(continued on next page)

```
% canvas.  This is a utility used by /singlerequest.  Note that the
% list is always empty if this is not a PrimarySelection.
%
/SelectedChildren { % - => [canvases]
        [ Rank /PrimarySelection eq {
                /children+ Holder send {
                        /Selected? 1 index send not {pop} if
                } forall
        } if ]
} def

% Subclass responsibility: The only requests we understand are (1)
% render the selection as an array of circles, (2) render it as a
% single canvas (the parent), and (3) delete the selected canvases
% from the CirclesCanvas parent.
%
% For the first, we offset the x/y of each circle according to DeltaX/Y.
% If those methods have not been promoted, this causes each circle to be
% given a 0/0 origin, which is then offset by the destination coords.
% If /SetDeltas has been called, DeltaX/Y offset the circles by the
% starting coordinates of the drag operation.
%

% The second form of query, /Canvas, isn't by any means required;
% indeed, there are few if any clients who query for such a value.  But
% it's so trivial to provide that most Selections might as well offer it.
% Note however that it is NOT included in the /CanRenderAs array, lest
% the /Canvas value get copied to the ClipBoard and thus prevent the
% window from going away in a timely fashion.
%
% For /DeleteContents, we use the trick of turning the to-be-deleted
% canvas(es) opaque just before deleting them, to force that portion
% of the parent to be repainted.  Other selection clients might be able
% to make more precise calculations of what needs to be repainted.
%
/singlerequest { % oldval key => newval
    {
        /Circles {
            pop [ SelectedChildren {
                dup /baggage Holder send aload pop
                /FillColor 5 -1 roll send
                colorrgb pop pop        % x y r gray
                4 -1 roll DeltaX sub
                4 -1 roll DeltaY sub
                4 2 roll 4 array astore
            } forall ]
```

(continued on next page)

```
            }
            /Canvas {
                pop Holder
            }
            /DeleteContents {
                SelectedChildren {
                    false /settransparent 2 index send % force damage
                    /removeclient Holder send { /destroy exch send } if

                } forall
            }
            /Default { pop /UnknownRequest }
        } case
    } def
    % Stash the starting coordinates of a drag operation.
    %
    /SetDeltas { % x y => -
            /DeltaY exch promote
            /DeltaX exch promote
    } def

classend def

% Finally, here's the code to fire up the windows.

/c1 CirclesCanvas nullarray framebuffer /newdefault ClassBaseFrame send def
(Selection Example) /setlabel c1 send
(SelDemo) /seticonlabel c1 send

/c2 CirclesCanvas nullarray framebuffer /newdefault ClassBaseFrame send def
(More Selections!) /setlabel c2 send
(SelDemo) /seticonlabel c2 send

/activate c1 send        /activate c2 send
/place c1 send           /place c2 send
/map c1 send             /map c2 send

% Because both windows were fired up from the same process, they share
% the userdict that contains /c1 and /c2.  Therefore neither window will go
% away until both have been told to Quit.  (When both processes have been
% killed, the userdict will go away.)  To avoid this problem, we tell
% each frame's EventMgr to put a copy of the frame on the stack, then get
% rid of the copies in userdict.

{c1 /c1 null def} /callmanager /EventMgr c1 send send
{c2 /c2 null def} /callmanager /EventMgr c2 send send
```

(continued on next page)

```
% This last part lets you run the code through psh and have it exit cleanly.

newprocessgroup
currentfile closefile
```

10. MISCELLANEOUS TOPICS

10. MISCELLANEOUS TOPICS

10 Miscellaneous Topics

Miscellaneous Topics 10-1
ClassTarget 10-1
- Setting and Getting the Target 10-1
- Sending to the Target 10-2
- Example 10-2
- Automatic Menu Targets 10-3
- Disappearing Targets 10-3
- How Targets Work 10-3

NeWS Development Environment Applications 10-4
- Taxonomy of Applications 10-4
- Starting an Application 10-4
- Killing an Application 10-6

Miscellaneous Topics

ClassTarget

ClassTarget is designed to allow application programmers to safely keep a reference to one object inside another object. Target references are 'safe' in the sense that they look after all the associated NeWS reference counting issues. (See the Memory Management chapter in the X11/NeWS reference manual for a full explanation of reference counting in the X11/NeWS server.)

You can instantiate ClassTarget directly, but applications will more commonly use this class as a mix-in. At present ClassTarget is mixed-in to ClassControl and ClassSelectionList, and hence its features can be used directly in any control or menu.

The reason that controls and menus include ClassTarget is because they both have callback procedures in which application programmers specify the action that should take place when a button is pressed, a slider is dragged, a menu item is selected, and so on. Typically this action will consist of sending a message to some *other* object. It is the target mechanism that maintains the reference to this other object.

Setting and Getting the Target

Sending **/settarget** to an instance of some subclass of ClassTarget associates a target object with it. The **/target** method returns this object. The argument to **/settarget** can be any NeWS object, although it will typically be an instance of some class. If your target is not an instance (or a class), the **/sendtarget** method (see below) will cause a a typecheck error.

It is not usually necessary to explicitly un-set a target. The act of destroying an object with a mixed-in target will automatically clear that target reference. A target can however be manually cleared by calling the method **/cleartarget**.

Methods:
/settarget **/cleartarget** **/target**

Sending to the Target

The target mechanism does not automatically dispatch your callback to some remote object. You must explicitly do this by calling **/sendtarget** from within your callback. If the target is currently null /sendtarget will cause an error.

Methods:
/sendtarget

Example

The following example assumes that we want to create an OpenLookButton whose purpose is to print something in the footer of a frame (not necessarily the one containing the button) when the button is pressed. Assume that we already have a frame called /myframe:

```
/mybutton
        (Press Me) {(Got it!) null /setfooter /sendtarget 5 -1 roll send}
        framebuffer /new OpenLookButton send
def

myframe /settarget mybutton send

% Insert mybutton into some bag, or simply
% activate it and map it onto the framebuffer, as in the
% /demo method for OpenLookButton
```

When mybutton is pressed the button itself will be put on the stack, and the callback executed. The '5 -1 roll' makes the button the subject of the /sendtarget method, which in turn makes myframe the subject of the **/setfooter** method. Finally the left-hand footer is set to the string "Got it!".

Note that **/settarget** is called after the button is created, but *before* it is activated. If the button were activated before the call to **/settarget** there would a chance that the user could press it while the target was still null. This would cause an error.

Note finally that since the callback does not contain a direct reference to any particular frame, there is nothing to prevent the application from changing the button's target (via **/settarget**) at any time. This would cause the /setfooter message to be sent to some other object the next time the user pressed the button. Of course, this new target object should be some other frame (or should at least understand the **/setfooter** message).

Automatic Menu Targets

In recognition of the fact that the obvious target for a menu callback is the canvas which received the MENU press and caused the menu to be shown, ClassCanvas includes code to automatically set the target for menus that it manages.

By default (and unless the automatic menu targeting is defeated) every time a menu is brought up the canvas which received the MENU press sets *itself* to be the target of the menu before showing it. See the sections on Menus and ClassCanvas for a more detailed explanation of this behavior.

Disappearing Targets

Targets do not ensure the continued existence of the objects they reference. If the application removes its last hard reference to the targetted object, the targetting mechanism will notice this and clear the (soft) target reference. Thus it is possible for a target to become null without having been explicitly cleared by the application. Robust applications which expect this behavior should not blindly use **/sendtarget** in their callbacks. They should first test whether there is still a target to send to.

How Targets Work

A target consits of two NeWs data structures: a soft reference to the remote object, and an interest in that object's obsolesence expressed in the global UI event manager.

When no more hard references to the targetted object exist NeWS sends an /Obsolete event signalling that its useful life has come to an end, and all remaining references (the soft ones) should be cleared so that storage allocated for the object can be reclaimed.

This /Obsolete event is caught not only by the object itself (through the standard mechanism in class Object), but also by the instance of ClassTarget which is referencing it. On receipt of this event the ClassTarget instance silently removes its reference to the targeted object, thus allowing it to be reclaimed.

NeWS Development Environment Applications

Taxonomy of Applications

tNt application programs fall into three natural categories, depending on their complexity:

1. PostScript-only applications. These are small programs, often demonstration programs, that execute entirely in the server. The code to implement these programs is typically downloaded via psh(1), and is usually places in the userdict associated with the psh connection.
2. PostScript / CPS applications. Any application with a client-side will use CPS, and those without asynchronous messages from the server to the client may not need to use the Wire Service.
3. PostScript / CPS / Wire Service applications. This is the most general category, and will include most medium and large scale applications. These applications typically start by sending PostScript code to the server via CPS. They then enter the Wire Service's Notifier, which receives and distributes the user interface events that require client-side processing.

Starting an Application

See the Wire Service sections of this manual and the CPS section in the NeWS 2.0 Programmer's Guide for details of starting an application with a client-side. What follows are the simple details of starting a PostScript-only application, or the PostScript-based component of a mixed client-server application.

Below is a standard template for starting the PostScript component of an tNt application. It is explained below.

```
% Definition of 'MyAppCanvas', and other
% application-specific classes.
% ...

/frame
    [MyAppCanvas] [] framebuffer /new OpenLookBaseFrame send
def

/activate frame send
/place frame send
/map frame send

% These two lines are only present in PS-only applications.
newprocessgroup
currentfile closefile
```

Most applications start by placing a base frame on the screen. The client of this frame will be an instance of some application-specific class.

It is important that a reference to this frame be maintained in some non-transient dictionary. Without this reference the frame and its contents would be reclaimed by the NeWS memory management system immediately. This is the reason for defining /frame in the userdict.

Typical applications then activate the frame, by sending the /activate method to it. This has the effect of starting event management on the frame and every canvas within it.

The next task is to establish a position and size for the application. /place performs this task. It consults the desktop manager for a position, obtains a size by sending the /preferredsize method to the frame.

Next the frame is made visible by sending the /map method to it. After this point the application is running independently, and responding to user input.

Finally, and only if the application is completely server-based and loaded via psh(1), the cliche "newprocessgroup currentfile closefile" should be executed. This has the effect of breaking the connection with the psh without killing the application.

Killing an Application

PostScript-only applications that were launched in the above manner may be killed in two ways:

- By the user selecting "Destroy" from the frame's menu. Applications can defeat this destruction by overriding the frame's /destroyfromuser method.
- By the application removing the reference to its base frame in the userdict. This will cause all the resources consumed by the application to be reclaimed by the NeWS server.

Applications with a client component can most simply kill themselves by breaking their connection. The most obvious way to do this is to exit their UNIX process.

11. INTERFACE REFERENCE

11. INTERFACE REFERENCE

11 Interface Reference

Interface Reference

Interface Reference 11-1
Introduction 11-1
Wire Service 11-1
- Error Handling 11-2
- Connection Management 11-2
- Handle Allocation and Registration 11-8
- Notifier 11-12
- Ease Of Use Macros 11-13
- Synchronization 11-14
- Constants 11-15

AbsoluteBag 11-15
- Direct Methods 11-15
- Class Variables 11-18

ClassBag 11-19
- Direct Methods 11-19
- Subclass Methods 11-26
- Class Variables 11-29

ClassBaseFrame 11-30
ClassButton 11-30
- Direct Methods 11-30
- Subclass Methods 11-31

ClassCanvas 11-31
- Direct Methods 11-32
- Subclass Methods 11-42
- Class Variables 11-47

ClassCommandFrame 11-48
ClassContainer 11-49
- Direct Methods 11-49
- Subclass Methods 11-51
- Class Variables 11-52

ClassControl 11-52
- Direct Methods 11-54
- Subclass Methods 11-58
- Class Variables 11-60

ClassDialControl 11-61
ClassFrame 11-61
ClassHelpFrame 11-62
ClassIconFrame 11-62
ClassMenu 11-62
ClassPropertyFrame 11-63
ClassSelectionList 11-63
ClassTarget 11-63
- Direct Methods 11-64
- Subclass Methods 11-65

ClassTextControl 11-65
- Direct Methods 11-66
- Subclass Methods 11-73
- Class Variables 11-75

FlexBag 11-76
- Direct Methods 11-77
- Utility Methods 11-80

Object 11-81
- Direct Methods 11-81
- Subclass Methods 11-85
- Class Variables 11-86

OpenLookAbbrButton 11-86
- Direct Methods 11-86

OpenLookAbbrButtonStack 11-90
OpenLookBaseFrame 11-90
- Direct Methods 11-91

OpenLookButton 11-102
- Direct Methods 11-102
- Class Variables 11-105

OpenLookButtonStack 11-106
- Direct Methods 11-106
- Subclass Methods 11-110

OpenLookCheckBox 11-110
OpenLookChoggle 11-111
OpenLookCommandFrame 11-111
OpenLookFrame 11-112
OpenLookHelpFrame 11-112
OpenLookHorizontalScrollbar 11-112

OpenLookHorizontalSlider 11-113
- Direct Methods 11-113
- Subclass Methods 11-114

OpenLookIconFrame 11-115
OpenLookMenu 11-116
- Direct Methods 11-116

OpenLookNonXSetting 11-122
- Direct Methods 11-122

OpenLookNoticeFrame 11-127
OpenLookNumeric 11-127
- Direct Methods 11-128
- Class Variables 11-133

OpenLookPane 11-134
OpenLookPropertyFrame 11-134
OpenLookTextControl 11-135
- Direct Methods 11-135
- Subclass Methods 11-142
- Class Variables 11-144

OpenLookVerticalScrollbar 11-145
- Direct Methods 11-145
- Subclass Methods 11-148

OpenLookVerticalSlider 11-148
OpenLookXSetting 11-149
- Direct Methods 11-149

Subclass Methods 11-154
OpenLookXSettingControl 11-154
RowColumnBag 11-154
- Direct Methods 11-155

RowColumnLayout 11-157
- Direct Methods 11-158
- Subclass Methods 11-159
- Class Variables 11-160

Interface Reference

Introduction

The following sections detail the programmer's interface to tNt. First, the Wire Service functions are described, then the method interface for selected classes on the server side of tNt.

Note that for the server side, the reference is incomplete. Only the most commonly used classes are described, and only selected methods are given for those classes.

Wire Service

The purpose of the NeWS Wire Service is to provide a server-client communications package of sufficient generality to support diverse client applications and toolkits.

The Wire Service is nearly independent of the PostScript language server components of tNt. It does not presume the existence of any particular class, and should work as well with Lite clients as those based on the new PostScript toolkit (with the exception of the synchronizatrion routines.) It is an extension to CPS.

The components of the Wire Service are:

- a connection manager to handle multiple connections to one or more servers;
- "handle" allocation procedures, so that items on one side of the wire may be referred to from the other side;
- a lightweight notifier on the C-side so that asynchronous messages from the server(s) can be dispatched to client functions;
- and a synchronization package so that server-based code can make RPC-style calls across the wire. These four components are described below, after the error conventions and reporting facilities.

Error Handling

Most Wire Service interface functions return a value which can be coerced to an integer and tested for a 0 return value. Many of them return a boolean: `TRUE` for sucess, `FALSE` for failure.

```
int     wire_Errno;

char *
wire_ErrorString();

void
wire_Perror(prefix)
char    *prefix;
```

When an error has occurred, its type is available in the wire_Errno global variable, and a descriptive string is pointed to by `wire_ErrorString`. Like its UNIX equivalent the error condition is not cleared immediately after an error. It remains set until the next error. The function `wire_Perror` prints the current error string to standard error, prefixed by the user-supplied string.

Connection Management

The first component of the wire Service is the connection management routines, which support multiple connections per server and multiple servers per application.

```
/* Start talking to a server */
        wire_Wire
        wire_Open(server)
        char    *server;

/* Stop talking to an opened connection */
        boolean
        wire_Close(w)
        wire_Wire       w;

/* Return a pointer to the psio input file pointer */
        PSFILE *
        wire_PSinput(w)
        wire_Wire       w;

/* Return a pointer to the psio output file pointer */
        PSFILE *
        wire_PSoutput(w)
        wire_Wire       w;
```

wire_Open takes an argument to specify the particular server to connect to. An argument of NULL will cause the the NEWSSERVER environment variable to be used. If there is no such environment variable, DISPLAY is used. If this also does not exist then the current host i s used with default port 2000. If the argument is not NULL then it should be a hostname, a NEWSSERVER-style string, or a DISPLAY-style string. These formats are discussed in the X11/NeWS documentation.

wire_Close is straight-forward. If the argument is wire_ALLWIRES, all of the connections will be closed. In this case, FALSE will be returned if there is an error with any of the connections.

wire_PSinput and wire_PSoutput are accessor functions to the psio file pointers. These are needed if a program wishes to access the psio files. (More details may be found in the *NeWS Programmer's Guide.*) Note that the current implementation uses 2 file descriptors per connection. Thus, the number of available wires is determined by the number of available file descriptors in the system. This is highly implementation-specific and may be changed in the future.

```
/* Direct CPS to and from a given connection from now on */
        boolean
        wire_SetCurrent(w)
        wire_Wire       w;

/* Report current connection */
        wire_Wire
        wire_Current()

/* Check the validty of wire w */
        boolean
        wire_Valid(w)
        wire_Wire       w;
```

`wire_SetCurrent` has the effect of moving the appropriate file pointers into the **PostScript** and **PostScriptInput** global variables. All libcps calls will thereafter use this connection. The act of opening a wire does **not** set it to be the current wire. This must be done manually.

`wire_Current` returns the current connection. It is necessary because the Notifier (see below) may itself change the current connection, depending on where the next message has come from. Clients that do not want to reply down the same connection as their up-coming message will have to call `wire_SetCurrent` again before they write.

wire_Valid returns `TRUE` if w is a valid wire, or `FALSE` if it is not.

```
/* Associate client data pointer with a connection */
        boolean
        wire_SetData(w, data)
        wire_Wire       w;
        caddr_t         data;

/* Return client data pointer for a connection */
        caddr_t
        wire_Data(w)
        wire_Wire       w;
```

Applications may associate client data with each connection via the `wire_SetData` and `wire_Data` interfaces. The most common use of this will be to reestablish some per-connection application context when processing a message from a particular connection.

```
/* Enable a connection */
        boolean
        wire_Enable(w)
        wire_Wire       w;

/* Disable a connection */
        boolean
        wire_Disable(w)
        wire_Wire       w;

/* See whether a connection is enabled */
        boolean
        wire_Enabled(w)
        wire_Wire       w;
```

`wire_Disable` is used to remove a wire from the Notifier temporarily. Later, `wire_Enable` can be used to restore Notifer service to the wire. While a connection is disabled, the Notifier will not read any messages from it, and no functions will be called on its behalf. The purpose of this function is to allow a client to negotiate with one server, and guarantee that it won't be interrupted by messages from another. When first opened, a connection is enabled. Disabling a wire only affects its input side; writes to a disabled wire will succeed. The function `wire_Enabled` reports whether a particular connection is currently enabled.

`wire_ALLWIRES` may be passed as a paramater to `wire_Disable` and `wire_Enable` in order to disable/enable all of the wires in a single call. An error is reported if there is a problem with any one of the connections.

```
/* Register functions to be called on
   connection death, reading error */
        wire_Problems(w, death, disease, unknowntag)
        wire_Wire       w;
        void            (*death)();
        void            (*disease)();
        void            (*unknowntag)();

        /* User function, abnormal connection termination */
                void
                (*death)(w)
                wire_Wire       w;

        /* User function,  connection protocol error */
                void
                (*disease)(w)
                wire_Wire       w;

        /* User function,  unregistered tag is found */
                void
                (*unknowntag)(w)
                wire_Wire       w;

        /* Default functions for wire problems */
                void    wire_DeathDefault();
                void    wire_DiseaseDefault();
                void    wire_UnknownTagDefault();

        /* Eat a token and skip to the next tag */
                boolean
                wire_SkipEvent()
```

Application programmers may supply three functions that the notifier will call after particular abnormal events. If the connection is terminated, other than by a call to `wire_Close,(*death) ()` is called. This user-supplied function should not attempt to close the offending wire. If the notifier finds a token at the head of an input queue that is not recognizable as a dispatching tag, `(*disease) ()` is called, and the current connection is preset to the offending one. It is the responsibility of this function to consume the leading non-tag values from the stream. Finally, if the notifier finds a dispatching tag which has not been registered using `wire_RegisterTag, (*unknowtag) ()` is called. A `NULL` argument to any of these three arguments to `Problems` will leave that function unchanged.

If wire_Problems is not called, the functions wire_DeathDefault, wire_DiseaseDefault and wire_UnknownTagDefault are used. wire_DeathDefault prints a message to stderr, DiseaseDefault cleans up the queue and prints a message to stderr, and wire_UnknownTagDefault eats the tag and any following arguments, also printing a message to stderr.

If wire_Problems is called with wire_ALLWIRES as the first parameter, then the same set of callbacks will used for all connections.

wire_SkipEvent consumes the initial token on the current wire and any remaining input upto, but not including the next tag. If there is no next tag on the current wire wire_SkipEvent will not block waiting for one. This function is useful when writing disease and unknowntag functions.

```
boolean
wire_AddFileHandler(file, callback, data)
FILE    *file;
void    (*callback)();
caddr_t data;

boolean
wire_RemoveFileHandler(file)
FILE    *file;
```

wire_AddFileHandler adds a file to the Notifier's list of files to check. When data is detected on the file, the callback is called and passed the data pointer. wire_RemoveFileHandler removes a file from the list. Note that the file is not a wire_Wire and cannot be enabled or disabled. There are no restrictions imposed on the file and it is up to the client to handle all operations within the callback.

Note that these routines all take a file pointer. If a file descriptor is desired in the application program, a call to fdopen can be made with no adverse side-effects.

```
/* Map a wire to a small integer */
        int
        wire_WireToInt(w)
        wire_Wire w;

/* Map a small interger into a wire */
        wire_Wire
        wire_IntToWire(i)
        int i;
```

Certain clients of the Wire Service may want to build data structures that are indexed by a wire. For this reason a pair of procedures (currently macros) are provided that map a wire into a unique small integer and back again. This is meant to be used in those cases where client does not want to use the client data field associated with the connection.

Handle Allocation and Registration

Both the C and PostScript language components need to reference remote objects. The C programmer may need to modify or query some PostScript language object he created earlier. Similarly, any PostScript language object which wishes to notify the client of a user event needs some way to specify the appropriate C function to invoke. Since references to PostScript language objects can not be passed across the wire, and C pointers can not easily be stored in the PostScript language world, we provide two "handle allocators" which generate and remember unique identifiers.

```
/* Reserve a range of client tags */
        int
        wire_AllocateTags(count)
        int     count;

/* Assign client tags to an array of addresses */
        boolean
        wire_AllocateNamedTags(names)
        char    *names[];

/* Make the tag allocator ignore a range of integers */
        boolean
        wire_ReserveTags(largest)
        int     largest;
```

The Wire Service uses "tags", as provided by CPS, to drive its notifier. Before you can register a callback with the notifier, you must obtain a tag to associate with the callback.

`wire_AllocateTags` takes a number N and returns another M, such that none of the integers M, M+1, ... M+N-1 are already allocated. These integers are handles whose primary use will be to dispatch messages from the server to client functions.

`wire_AllocateNamedTags` is a thin wrapper around `wire_AllocateTags`. It takes a `NULL` terminated array of pointers to integers, and assigns a tag through each of these pointers. Here is a typical use:

```
int     menu_tag, resize_tag;
int     *tag_pointers[] = {&menu_tag, &resize_tag, NULL};

wire_AllocateNamedTags(tag_pointers);
wire_RegisterTag(menu_tag, my_menu_callback, data);
wire_RegisterTag(resize_tag, my_resize_callback, data);
        ...
```

`wire_ReserveTags` is provided to allow dynamically-allocated tags to coexist with old-style constant tags. If you know that some piece of code uses tag values 1..50, then before calling `AllocateTags` you should call `wire_ReserveTags(50)`. This facility can also be used to leave space for your own private tag allocator if the one provided by the Wire Service doesn't meet your needs. Note: `ReserveTags` must be called before any connections are opened.

```
/* Register client handler */
        boolean
        wire_RegisterTag(tag, funcp, data)
        int     tag;
        void    (*funcp)();
        caddr_t data;

        /* The notifier will call your
           function in the following way */
                void
                (*funcp)(tag, data)
                int     tag;
                caddr_t data;

/* Retrieve client function pointer */
        void (*)()
        wire_TagFunction(tag)
        int     tag;

/* Retrieve client data pointer */
        caddr_t
        wire_TagData(tag)
        int     tag;
```

`wire_RegisterTag` allows you to associate a function pointer and a user data pointer with a tag. If this tag is ever found on the wire by the notifier, your function will be called. `TagFunction` and `TagData` retrieve the previously registered information.

The Wire Service uses `CPS` usertokens as handles to PostScript language objects. These tokens are allocated on a per connection basis. The application is reponsible for the registration of the usertoken in the server. These three calls are analogous to the above calls for tag allocation, except that they are done on a per-wire basis. Unlike `wire_ReserveTags`, `wire_ReserveTokens` is called after the connection has been opened.

```
/* Reserve a range of client token on
        a specific wire */

        int
        wire_AllocateTokens(w, count)
        wire_Wire       w;
        int             count;

/* Assign tokens for a specific wire to
        an array of addresses */

        boolean
        wire_AllocateNamedTokens(w, tag_array)
        wire_Wire       w;
        char            *names[];

/* Make the token allocator a specific wire
        ignore a range of integers */

        boolean
        wire_ReserveTokens(w, largest)
        wire_Wire       w;
        int             largest;
```

There is no equivalent to `wire_RegisterTag()` because this cannot be done from the C process – you need a reference to the PostScript language object to register it. Here is how you should use the usertoken facility for registering your server-side objects:

```
        In your C file:
                window_token = wire_AllocateTokens(1);
                ps_CreateMyWindow(window_token);

        And in your CPS file:
                cdef ps_CreateMyWindow(int token)
                        ... /new MyWindow send          % win
                        token setfileinputtoken
```

Notifier

The purpose of the Notifier is to read tags from one or more server connections, and depending on their value, call particular client functions. The functions that are called are those that were previously registered using `RegisterTag()`. The Wire Service provides both popular styles of notification: the notifier itself can handle the main loop, or else the client program can repeatedly request the dispatching of a single incoming message. The two styles can also be mixed in the same application.

```
/* Read, process one message */
        boolean
        wire_Notify(timeout)
        struct timeval  *timeout;

/* Any messages to read on this (or any) wire? */
        boolean
        wire_WouldNotify(w)
        wire_Wire       w;

/* Descend into main loop */
        void
        wire_EnterNotifier()

/* Emerge from main loop */
        void
        wire_ExitNotifier()
```

`wire_Notify` causes a single tag to be read from one of the active connections. (Round-robin scheduling is used when there is more than one connection with data ready for reading.) This tag is used as an index into the table of registered procedures. The procedure is then called with the handle and registered data as arguments. (See the function my_slider_handler in the example below). `wire_Notify` has the side-effect of setting the current connection, so that registered functions can read further arguments from the wire using normal `CPS` and psio functions. If there is no data available on any of the active connections, `wire_Notify` will block until some message arrives or the period specified in the timeout parameter expires. If a timeout occurs or the block is interrupted by a system call, `wire_Notify` will return an error.

wire_WouldNotify does not block, and reports whether there are any pending messages on the specified wire. The special argument, wire_ALLWIRES, causes this procedure to return TRUE if any of the active connections have input.

wire_EnterNotifier will be the main-loop for many client applications. It is reentrant, and can be intermixed with calls to wire_Notify. In fact, it will do little more than repeatedly call wire_Notify itself. A call to wire_EnterNotifier will not return until the corresponding wire_ExitNotifier has been executed.

wire_ExitNotifier is called when the application programmer wants to exit from a (possibly nested) notifier loop. The corresponding wire_EnterNotifier will return as soon as the registered procedure which called wire_ExitNotifier itself returns. Pending messages are not processed in any way.

Ease Of Use Macros

The following macros are provided to enable data to be easily read from the current connection. It is assumed that the user of these macros knows the type of the data on the wire. Thus there is no type checking or error reporting. If the data is of the wrong type, garbage may be returned and the wire may be left in an undetermined state. The caveat to this is that numeric arguments are converted by CPS (floating to integer and vice versa).

```
int     wire_ReadTag()
int     wire_ReadInt()
float   wire_ReadFloat()

char    *
wire_ReadString(str)
char *str;

void    wire_GobbleAny()
```

Synchronization

CPS provides a mechanism for a client process to block pending notification from a server process. The wire service provides a complementary mechanism which will allow a server process to block pending notification from a client process. This will provide symmetric facilities for synchronous communications.

The server interface looks like:

```
proc /wire_Sync => --
```

The *proc* is executed, and wire_Sync guarantees it will not return until the C client has acknowledged dealing with anything sent to it by the proc. Thus, for example, the PostScript program can ask the C program to send it some value, or to do some painting, etc., and be sure that C has responded to the request before trying to do any more PostScript language code. Naturally, this makes some assumptions about the client; hence the requirement that the client be built on the Wire Service.

The client interface is:

```
/* Returns TRUE if the wire is responding to
        a synchronised request */

        boolean
        wire_InSync(w)
        wire_Wire w;
```

wire_InSync simply checks to if the specified wire is responding to a synchronised request from the server, which means that PostScript language code sent now may be executed before PostScript language code sent earlier. This is a subtle but important point.

Constants

The following constants are defined in the wire service:

```
Boolean values: TRUE    FALSE

Special Wires:  wire_INVALID_WIRE       wire_ALLWIRES

Error values:
                    wire_EUNKNOWNHOST       wire_ENOSUCHSERVER
                    wire_EBADWIRE           wire_ECONNECTIONDIED
                    wire_ENOWIRES           wire_ETIMEOUT
                    wire_EINTR              wire_ERANGECHECK
                    wire_EBADRESERVE        wire_EFILEINUSE
                    wire_ENOFILEHANDLER     wire_ECONNECTIONREFUSED
```

AbsoluteBag

Subclass of **ClassBag**
Source file: **bagutils.ps**

This class can be directly instantiated

An AbsoluteBag is a general purpose bag in which clients are given an absolute position when they are inserted. Regardless of the size and shape of the bag, clients always remain in this position thereafter.

Direct Methods

`/addclient`

`name|null [x y client]` **/addclient** –
`name|null [x y client_class_args client_class]` **/addclient** –

There are several parameters required to add a client to an absolute bag:

- name, the client may be named or unnamed (**null** as the first argument),

- baggage, which must be present in an absolute bag. It consists of the *x* and *y* coordinates at which the client will be located.
- client, which may be an instance or a class to instantiate, along the same lines as described in the section on ClassBag.

In the example below, the bag makes an instance of OpenLookButton, and stores it in the bag with the client name supplied.

```
/b1 [20 50 (Foo) {...} OpenLookButton] /addclient mybag send
```

See the *ClassBag* section for details on the naming and instantiation of bag clients.

`/baggage`

`client` **/baggage** `[x y]`
Returns the baggage, which specifies the absolute position of the client in the bag.

`/clientcount`

– **/clientcount** `n`
Returns the number of clients currently in the bag.

`/clientlist`

– **/clientlist** `[client1 client2 ...]`
Returns an array of clients, with the canvas clients in the same order as they were inserted into the bag, unless explicitly changed by the application.

`/destroy`

– **/destroy** –
Destroy the bag and its clients. Refer to the ClassBag **/destroy** method for the additional information on the use of this method.

`/location`

– **/location** `x y`
Return the location of the origin of the bag relative to the CTM.

`/minsize`

/minsize `minwidth minheight`
Compute the minimum acceptable size for this bag, This is how big the bag must be to hold the clients at their current sizes (not their minumum

sizes). A well behaved application will respect this size when reshaping the bag in response to user mouse actions.

/move

`x y` **/move** –
Move the origin of the bag to the specified location, in CTM coordinates.

/new

`parentcanvas` **/new** `instance`
Create an absolute bag parented to the specified canvas.

/preferredsize

– **/preferredsize** `preferredwidth preferredheight`
Calculate the "ideal" size of the bag, which defaults to the minimum size. Well behaved applications will respect this size when initially displayed the bag.

/removeclient

`client|name|n` **/removeclient** `oldclient true`
`client|name|n` **/removeclient** `false`
Remove the client given, named or indexed in the argument. The method returns **true** and the client object if the client is found, otherwise it returns **false**.

/reshape

`x y w h` **/reshape** –
Reshape the bag to the dimensions given and invalidate it. This results in the bag being layed out as the first step in painting it.

/sendclient

`<args> /method /name` **/sendclient** `results`
Send the specified method, with any arguments, to the named client. An error results if the client is not in the bag.

/setbaggage

`client [x y]` **/setbaggage** –
Set the client's positioning coordinates.

`/settopdown`

`bool` **/settopdown** –
Define the orientation of the bag's coordinate system as seen by clients: **true** sets the coordinates "top-down', **false** sets them "bottom-up". See the **/topdown?** method for more details.

`/size`

– **/size** `w h`
Return the width and height of the bag in coordinates of the CTM.

`/topdown?`

– **/topdown?** `bool`
Return the positioning of the bag's coordinate system: true if the coordinates are "top-down', with (0,0) at the upper left corner of the bag. False means the coordinates are "bottom-up", with (0,0) at the bottom left corner, which is the standard NeWS orientation.

Note first, that this positioning effects the clients of the bag as well as the bag itself. If the bag's coordinates are "top-down" (origin at the upper left), then so is the origin of each client.

Note second, that this variable only effects the orientation of clients inside the bag. Externally, as a NeWS canvas, the bag's origin remains at the lower left corner, regardless of this variable.

Class Variables

`/TopDown?`

This boolean variable reflects the positioning of the bag's coordinate system: **true** if the coordinates are "top-down', `false` if the coordinates are "bottom-up". See the **/topdown?** method for more details.

see also:

ClassBag

ClassBag

Subclass of **ClassCanvas**
Source file: **bag.ps**

This class should be subclassed rather than directly instantiated.

A bag is a type of canvas that manages a group of clients, either canvases or graphics. Bags formalize the concept of containment.

When a client canvas is added to a bag, it is reparented to the bag, becoming a child of the bag in the NeWS canvas tree. Thus, in most applications, the interior canvases descended from the application's base frame are typically bags.

A bag provides its clients with names, shared event management, and collective layout and damage repair.

Direct Methods

`/activate`

– **/activate** –
Activate event management for the bag and all its canvas clients. This method is called recursively on clients that are themselves bags.

`/addclient`

`name|null [... client]` **/addclient** –
`name|null [... clientclass]` **/addclient** –
`name|null client` **/addclient** –
There are several ways to add a client to a bag.

The client can be specified as an instance of a subclass of either ClassCanvas or ClassGraphic. Or the client can be specified by a subclass of ClassCanvas, in which case, the bag will instantiate the class and accept that instance as its client. If arguments are required to instantiate the class, they must be supplied along with the "baggage" described below. Note that a parent argument is not required when specifying a class to instantiate: the bag will be the parent.

The client may be named or unnamed. The name may be any legal dictionary key in the PostScript language. If a name is given, the client can later be accessed by supplying that name to the **/getbyname** method. If **null** is given as the name, the client is accessed by the order in which it was put into the bag, the first client being number 0.

The application must remember the insertion order of unnamed clients. The bag will return a client given its index, but there is no method for which the bag returns the index.

The client can be accompanied by "baggage", which is data bundled into an array with the client instance (or class). This information is used by bag subclasses, typically in laying out the clients. The arguments necessary to instantiate a client from a class are supplied next to the baggage, but are not consumed during instantiation and are not stored.

Note that if you move a client from one bag to another, you must first send **/removeclient** to the bag the client is leaving before sending **/addclient** to the bag the client is entering.

For examples of adding a client to a bag, see the introductory section, *Bags*.

`/addfocusdescendant`

`client` **/addfocusdescendant** –

This method is used to notify all parent bags of this bag that they have a new descendant that is interested in input focus. This can happen either because a new key consumer was added to this bag or because an existing client changed status, becoming interested.

The method is called automatically when adding a client to a bag. It is not generally necessary to override this method, either.

`/addkeyconsumer`

`client` **/addkeyconsumer** –
This method is called automatically when adding to the bag a client that is interested in keyboard input (a key consumer).

`/baggage`

`client` **/baggage** `[data]`
Returns the baggage data that was passed in when *client* was added to the bag.

`/client?`

/name **/client?** `boolean`
Returns **true** if *name* belongs to an existing client in the bag.

`/clientcount`

– **/clientcount** n
Returns the number of clients currently in the bag.

`/clientlist`

– **/clientlist** `[client1 client2 ...]`
Returns an array of clients, with the canvas clients in the same order as they were inserted into the bag, unless explicitly changed by the application.

`/deactivate`

– **/deactivate** –
Turn off event management for the bag and all clients. This removes all interests expressed by the bag and clients. It also kills the event manager process if it is owned by the bag, otherwise it notifies the event manager that the bag is no longer active.

`/destroy`

– **/destroy** –
Destroy the bag and its clients; send **/destroy-dependent** to each client and remove any associated baggage. This method is called automatically when the NeWS memory reference count on the bag goes to zero.

Any clients remaining after sending **/destroy-dependent** are reparented to an offscreen canvas. This allows a subclasser to override **/destroydependent** for any client that should not be

destroyed as the result of destroying the bag and other clients.

`/destroydependent`

– **/destroydependent** –
This method is sent to each client of a bag when the bag is destroyed. By default, it does **/destroy self send.** Override this method for any client that should not be destroyed at the same time as the bag and other clients.

`/focustarget`

– **/focustarget** `canvas | null`
Return the current input focus target for the bag, or null if there is none. The focus target for a bag is the client canvas to which the bag will forward input focus when the bag receives it.

`/foreachclient`

`proc` **/foreachclient** –
Call the given procedure for each client in the bag, with the client on the top of the operand stack.

`/getbyname`

`name` **/getbyname** `client true`
`name` **/getbyname** `false`
Return the client named by the argument and **true** if the client exists in the bag. Otherwise return **false.**

Note that the name may be an integer if the client was entered without a name.

`/graphicclientcount`

– **/graphicclientcount** n
Return the number of graphic clients in the bag. The total number of clients (**/clientcount**) minus this number equals the number of canvas clients.

`/graphicclientlist`

– **/graphicclientlist** `[client1 client2 ...]`
Return an array of graphic clients of the bag, or the nullarray if there are none. To get an array of canvas clients, use **/children+.**

`/?invalidate`

`methodname` **/?invalidate** –
This method is used to control which methods invalidate the bag; a method will call **/?invalidate** with its own name as the argument and

/?invalidate will decide whether to invalidate the bag. By default the following methods call **/?invalidate: /addclient, /removeclient, /reshape.** Override this method if you want a bag not to invalidate automatically after one of those methods listed above.

/invalidate

– **/invalidate** –
This method is called by the toolkit (through **/?invalidate**) to mark the bag invalid, meaning that its layout is out of date. The application does not normally call this method directly, but instead it is indirectly invoked by **/addclient, /removeclient, /reshape.**

/lastfocustime

– **/lastfocustime** `time`
Return the time when the bag last had the input focus.

/layout

– **/layout** –
Perform layout on all clients of the bag to move and/or reshape them based on the current size and shape of the bag. Note that this method is not usually directly called by an application, but is invoked indirectly by **/addclient, /removeclient, /reshape.**

/location

– **/location** `x y`
Return the location of the bag's origin in CTM coordinates.

/minsize

/minsize `minwidth minheight`
Compute the minimum acceptable size for this bag. A well behaved application will respect this size when reshaping the bag in response to user mouse actions.

When subclassing, override this method if the calculations do not require the current canvas to be the bag. (If the calculations do require the bag as the current canvas, override **/MinSize** instead.)

/move

x y /move –
Place the origin of the bag at the point (x,y) in the coordinates of the CTM.

/new

parentcanvas /new instance
Create a bag parented to the specified canvas.

/newinit

– /newinit –
Override this method to initialize a bag with arguments supplied to **/new**. The overridden **/newinit** should use the arguments in its initialization and consume them (remove them from the stack).

/preferredsize

– /preferredsize preferredwidth preferredheight
Calculate the "ideal" size of the bag. Well behaved applications will respect this size when initially displayed the bag.

When subclassing, override this method if the calculations do not require the bag to be the current canvas. (If the calculations do require the bag as the current canvas, override /PreferredSize instead.)

/removeclient

client|name|n /removeclient oldclient true
client|name|n /removeclient false
Remove the client given, named or indexed in the argument. The method returns **true** and the client object if the client is found, otherwise it returns **false**. The effect is to remove the client and any of its baggage, reparent the client to an offscreen canvas, and deactivate it if it was using the bag's event manager. If the client was activated before being added to the bag (had its own event manager) it remains active.

/removefocusdescendant

client /removefocusdescendant –
This method is sent to notify a bag that a descendant (canvas) client has stopped being interested in input focus. If that descendant was

the focus target, this method calls **/MakeFocus-Target** to obtain a new focus target. This method is used a bag (**/removekeyconsumer**) to notify any parent bags when a child stops consuming keys.

`/removekeyconsumer`

`client` **/removekeyconsumer** –
Notify parent bags that a child of this bag has stopped being a key consumer. This can happen either because the child was removed from the bag, or remained in the bag but revoked its interests in keyboard focus events. This method is not directly called by an application, but is invoked through **/removeclient**.

`/reshape`

`x y w h` **/reshape** –
Reshape the bag to the dimensions given and invalidate it. This results in the bag being layed out as the first step in painting it.

`/sendclient`

`<args> /method /name` **/sendclient** `results`
Send the given method with arguments to the named client.

`/setbaggage`

`client [attributes]` **/setbaggage** –
Store the extra information ("baggage") for the client. Typically in bags this information is used in layout.

`/setfocustarget`

`clientcanvas|null` **/setfocustarget** –
The argument is stored in the **/FocusTarget** variable. If it is **null**, the bag will cease to be interested in input focus. Otherwise, the client canvas named in the argument becomes the focus target for the bag.

`/size`

– **/size** `w h`
Return the current size of the bag in the coordinates of the CTM.

`/valid?`

– **/valid?** `boolean`
Return **true** if the bag's layout is valid.

`/?validate`

– **/?validate** –
Validate if the bag's layout is currently invalid.

`/validate`

– **/validate** –
Lay out the bag in preparation for painting. This method is not generally called explicitly by an application, but rather gets invoked through **/paint** or **/fix**.

Subclass Methods

`/BagBegin`

– **/BagBegin** –
Set the bag as the current canvas and save the current NeWS graphic context, establishing a "bag context". This allows layout procedures to operate in the CTM of the bag.

`/BagEnd`

– **/BagEnd** –
Restore the previous NeWS graphics context.

`/CheapIntegerClient`

`int` **/CheapIntegerClient** `client`
Return the client corresponding to the array index given as an argument.

`/CheapIntegerName?`

`int` **/CheapIntegerName?** `boolean`
Returns **true** if the argument is an integer and clients are stored by number in an array.

`/DeRegisterByName`

`/name` **/DeRegisterByName** `oldclient true`
`/name` **/DeRegisterByName** `false`
Remove the named client from the bag. Return the named client and **true** if the client is in the bag, otherwise, return **false**.

`/DeRegisterByOrder`

`client|int` **/DeRegisterByOrder** `client true`
`client|int` **/DeRegisterByOrder** `false`
Remove the client indicated by the argument, either the client object itself or its index in the internal client array. The method returns the client object and **true** if it was present, otherwise **false**.

/DeRegisterClient	`/name|client` **/DeRegisterClient** `oldclient true`
	`/name|client` **/DeRegisterClient** `false`
	Remove a client, whether stored by name or by order of insertion.

/FixChildren	– **/FixChildren** – This method sends /fix to each canvas, not graphic, client of the bag
/Layout	– **/Layout** – Override this method to perform specific layout functions for a subclass, such as placement and sizing relative to the bag. When this method is called by /layout, the bag will be the current canvas, so the subclass can use information about the bag's size and coordinates.
/MakeFocusTarget	`newcanvas	null` **/MakeFocusTarget** `newcanvas'	null` Override this method to decide which client should receive the input focus. This method is called when adding a key consumer to a bag that has no focus target or when removing the current focus target.
/MinSize	– **/MinSize** `w h` This method is called by /minsize with the bag as the current canvas to calculate the minimum size for the bag. Override it to implement special calculations for a subclass, for example basing the minimum size of the bag on the minimum sizes of clients.
/NewClient	`parentcanvas` **/NewClient** `instance` This method is used when creating a bag client from a *clientclass* given as an argument to **/addclient**. It is executed in the context of the class being instantiated. Override this method to provide special treatment of a newly created client instance.

`/NoticeDescendantFocus`

event **/NoticeDescendantFocus** –
This method is called by **/NoticeFocus** when the input focus has moved into a child of the bag; it sets the focus target variable (**/FocusTarget**) to the child canvas.

`/NoticeFocus`

event **/NoticeFocus** –
This method is the callback from the interest expressed by the bag in noticing input focus changes; it is called when input focus enters a focus-forwarding bag or any of its children. It determines the focus target and transfers the focus to it via **/TransferFocus**. If the focus changes on the bag itself, **/NoticeSelfFocus** is called. If the focus changes on a client of the bag, **/NoticeDescendantFocus** is called.

`/NoticeFocusEnterExit`

event **/NoticeFocusEnterExit** –
This method is called when focus enters or leaves a bag hierarchy; not when focus shifts from one client to another client of a bag. Override it when special behavior is required on gaining or losing input focus.

`/NoticeSelfFocus`

event **/NoticeSelfFocus** –
This method is called when focus moves into or out of a bag. It transfers the focus to the client canvas indicated by **/FocusTarget**.

`/PaintChildren`

– **/PaintChildren** –
This method is called by /paint to paint each canvas client of the bag.

`/PaintGraphicChildren`

– **/PaintGraphicChildren** –
This method is called by /paint to paint each graphic client of the bag.

`/PreferredSize`

/PreferredSize **w h**
This method is called by **/preferredsize** with the bag as the current canvas. By default it returns **/minsize**. Override it to perform any special calculations for a subclass, for example, considering the preferred sizes of clients.

/RegisterByName	`/name client` **/RegisterByName** – This method adds a client to the bag by name.
/RegisterByOrder	`client` **/RegisterByOrder** – This method adds a client to the bag by the order of insertion.
/RegisterClient	`/name|null client` **/RegisterClient** – This method adds a client to the bag by name or by the order of insertion, depending on the argument given.
/TransferFocus	`canvas` **/TransferFocus** – Transfer input focus to the client indicated by the /FocusTarget variable.

Class Variables

/FocusForwarder?	This variable reflects the bag's interest in input focus on behalf of its clients. When **true**, this variable indicates that the bag will receive input focus and then pass it on ("forwward" it) to the client indicated by /FocusTarget.
/FocusNoticer?	This variable is **true** when the bag is interested in being notified whenever input focus enters or leaves it or any of its descendants.
/FocusTarget	The client stored in this variable receives the input focus if the /FocusForwarder? variable is **true**.

see also:

ClassCanvas

ClassBaseFrame

Subclass of **ClassFrame**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

This class is not for direct use; rather it supports frame (window) subclasses, such as the OPEN LOOK window frames. If you need a simple window frame for immediate use, consider one of the OPEN LOOK frames:

- OpenLookBaseFrame
- OpenLookPropertyFrame
- OpenLookNoticeFrame
- OpenLookCommandFrame
- OpenLookHelpFrame

ClassButton

Subclass of **ClassControl**
Source file: **button.ps**

This class should be subclassed rather than directly instantiated.

This class is the superclass for OpenLookButton as well as various pins and anchor boxes used throughout the toolkit. The following methods should be studied in conjuction with ClassControl.

Direct Methods

`/graphic` — `-` **/graphic** `obj`
Return the graphic object of the button.

`/setgraphic` — `thing|graphic` **/setgraphic** `-`
Convert the argument to a graphic, if necessary, and store it as the button's graphic. Then, invalidate the button to require layout before repainting.

Subclass Methods

`/CreateGraphic`

`thing` **/CreateGraphic** `graphic`
This method is SubClassResponsibility.

Override it to create a graphic from the *thing* supplied.

`/EnGraphic`

`thing|graphic` **/EnGraphic** `graphic`
Return a graphic corresponding to the argument supplied. If the argument is a terminal graphic, return it as is. Otherwise, use /CreateGraphic to make it into a graphic.

`/UnGraphic`

`graphic` **/UnGraphic** `thing|graphic`
Return the thing or graphic that was supplied to construct this graphic. If the object is already a terminal graphic, return it as is.

see also:

ClassControl

ClassCanvas

Subclass of **Object**
Source file: **canvas.ps**

This class can be directly instantiated, but is usually subclassed.

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

Direct Methods

`/activate`

– **/activate** –
Activate event management for this canvas; express the interests returned by the **/makeinterests** method. Create an event manager for the canvas if one cannot be found in one of its parent canvases.

`/active?`

– **/active?** `bool`
Return **true** if event management has been activated on the canvas.

`/autotargetmenu`

`bool` **/autotargetmenu** –
When **true** (the default), the canvas is automatically made the target for its associated menu whenever the menu is popped up.

`/autotargetmenu?`

– **/autotargetmenu?** `bool`
Return **true** if the canvas is automatically the target of its associated menu when the menu is popped up

`/bbox`

– **/bbox** `x y w h`
Return a bounding box for the canvas computed by the NeWS **getbbox** operator.

`/bboxfromuser`

– **/bboxfromuser** `x y w h`
Return a bounding box determined by mouse movement from the user. This method causes the cursor on the screen to change to a crosshair, and the NeWS process to block waiting for the user to press down the SELECT button. The user presses and drags out a rectangle that becomes the bounding box when the button is released.

`/callhelp`

– **/callhelp** –
Call the help procedure for this canvas.

`/canvas`

– **/canvas** `canvas`
This method returns the NeWS canvas object associated with this tNt canvas instance.

`/children+`

– **/children+** `array`
Return an array of NeWS canvases, the children of this canvas in the NeWS canvas tree. The array is ordered from the bottom canvas first to the top canvas last.

`/children-`

– **/children-** `array`
Return an array of NeWS canvases, the children of this canvas in the NeWS canvas tree. The array is ordered top canvas first to bottom canvas last.

`/deactivate`

– **/deactivate** –
Turn off event management for this canvas. If the canvas's event manager is shared with another canvas, notify the event manager to **/deactivatecanvas** for this canvas. If the event manager is owned by this canvas, destroy it.

`/descendants`

– **/descendants** `array`
Return an array of canvases, the descendants (if any) of this canvas. The array is ordered by proximity to this canvas, children before grandchildren. Within each generation, top canvases come before bottom canvases.

`/destroy`

– **/destroy** –
Destroy this canvas, have it removed from the canvas tree and interest lists. Before the NeWS server can reclaim the memory used by the canvas, all references to it must be remove. When subclassing, override this method to make sure any additional references you have made to an instance of the subclass are removed.

`/eventmgr`

– **/eventmgr** `eventmgr`
Return the event manager process for this canvas.

`/farthestcorner`

`x y` **/farthestcorner** `x' y'`
Return the corner of the canvas farthest from the given point, in the coordinates of the CTM.

`/fix`	– **/fix** – Called by the damage handler to repaint only the damaged area of a canvas using **/FixCanvas**. To paint the entire canvas use **/paint**.
`/getcolors`	– **/getcolors** `strokecolor fillcolor textcolor` Return the three colors of a canvas, used for stroking its outline, filling the interior, and text. By default these colors are those of the parent canvas.
`/help`	– **/help** `proc|null` Return the canvas's help procedure (as set by **/sethelp**), or **null** if there is none.
`/invalidate`	– **/invalidate** – Mark the canvas "invalid", the interpretation of which is left to subclasses. In the case of bags, for example, invalid means that the bag's layout is out of date.
`/isinside?`	`x0 y0 x1 y1` **/isinside?** `bool` Return **true** if the canvas is inside the box specified by its lower left and upper right corners respectively.
`/keyconsumer?`	– **/keyconsumer?** `bool` Return **true** if the canvas is interested in input focus.
`/lastfocustime`	– **/lastfocustime** `time` Return the last time the input focus was on this canvas.
`/location`	– **/location** `x y` Return the location of the origin of the canvas relative to the CTM.
`/lockminsize`	`null	w h` **/lockminsize** – Change the minimum size (**/minsize**) for this canvas instance. If the argument is **null**, restore the default value for the class.

`/makeinterests`

– **/makeinterests** `array`
Return an array containing the interests of this canvas, using the **/MakeInterests** method. In normal use, this method is nor directly called by an application.

`/map`

– **/map** –
Set the /Mapped attribute of the canvas. Refer to the *NeWS 2.0 Programmer's Guide* for the model of how this affects the canvas becoming visible on the screen.

`/mapped?`

– **/mapped?** `bool`
Return **true** if the canvas is currently mapped onto the screen.

`/menu`

– **/menu** `menu|null`
Return the menu object associated with this canvas as set by **/setmenu** and stored in the **/CanvasMenu** variable The menu is displayed whenever the MENU button is pressed with the pointer over the canvas. Return **null** if there is no menu set.

`/methodancestors`

`method` **/methodancestors** `array`
Return an array of the ancestors in the NeWS canvas tree that recognize the method supplied as an argument. If the class to which the method is sent recognizes the method, it is included in the array. If the sender of this method does not want the immediate recipient of the method included, it should send the method to the parent.

`/minsize`

– **/minsize** `w h`
Return the minimum size for this canvas as specified by the subclass of ClassCanvas to which it belongs. This value is used by the utility functions **/reshapefromuser** and **/stretchcorner**, which will not allow any user mouse action to make the canvas smaller than

/minsize, the smallest that the user can make the canvas with the mouse.

/move

`x y` **/move** –
Move the origin of the canvas to the specified location in the coordinates of the CTM.

/movefromuser

– **/movefromuser** –
Interactively move the canvas. This method uses a class variable **/DragFrame?** to control whether or not a wire frame or the canvas itself is shown as the canvas is moved.

/new

`parentcanvas` **/new** `instance`
Return a new canvas instance, with the specified canvas as its parent. The parentcanvas argument is consumed by **/newobject**.

/newinit

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

/paint

– **/paint** –
Paint the entire canvas, using the **/PaintCanvas** method.

/parent

– **/parent** `instance`
Return the NeWS canvas that is this canvas's parent in the canvas tree.

/parentdescendant

`canvas1` **/parentdescendant** `canvas2 true`
`canvas1` **/parentdescendant** `false`
Note that this method is sent to a class, not an instance.
This method asks a class to return the canvas closest to the given canvas in the NeWs canvas

tree that is also an instance of a subclass of that class. (Basically, it asks a class "which is the canvas closest to me that is also a descendant of you?") For example,

```
mycanvas /parentdescendant ClassFrame send
```

returns the frame surrounding *mycanvas* and **true**. If there is no such canvas, the method returns only **false**.

`/parents`

`-` **/parents** `array`
Return an array of the canvas's ancestors ordered from its (immediate) parent first to framebuffer last.

`/path`

`x y w h` **/path** `-`
Using the arguments to define a bounding box, this method returns a path that is the shape of the canvas sized to fit the box. You should override this method for any subclass that is not rectangular in shape.

`/place`

`-` **/place** `-`
This method is overridden by subclasses that have a default placement scheme. For example, frames override this method to specify their default placement on the framebuffer. OPEN LOOK frames override it to do staggered layout. The default action is to reshape the canvas from user input (**/reshapefromuser**).

`/preferredsize`

`-` **/preferredsize** `width height`
This method allows a subclasser to specify the size a canvas should be when it is started. (For example, the **/place** method in frames calls **/preferredsize** on its client canvas.) By default, this method simply calls **/minsize**. You should override this method (or **/PreferredSize**) to specify the 'ideal' size for a canvas subclass.

`/reparent`

`newparentcanvas` **/reparent** `-`
Move the canvas in the NeWS canvas tree, to have the given canvas as its parent.

Note: do not call this method when using bags; they handle reparenting themselves.

`/reshape`

`x y w h` **/reshape** –
Reshape the canvas to fit the bounding box specified by the arguments, in the coordinates of the CTM. This method allows the toolkit or the application to position the canvas on its parent.

`/reshapefromuser`

– **/reshapefromuser** –
Reshape the canvas from user interaction. First, an application sends this method to a canvas. Next, the cursor on the canvas changes to a crosshair, and the application's NeWS process blocks until the user performs a press-drag-release with the SELECT button. Then, the canvas is then resized to fit the resulting bounding box. but no smaller than the minimum size (**/minsize**).

`/scroll`

`dx dy` **/scroll** –
Scroll the contents of the canvas by the given increments. Non-retained canvases should override **/PaintScrolledArea** to update the canvas.

`/setcolors`

`strokecolor fillcolor`
`textcolor` **/setcolors** –
Sets the colors for stroking the border of the canvas, filling its interior, and drawing text. Null as an argument means do not change that value.

`/setcursor`

`int|kwd int|kwd|null` **/setcursor** –
`cursorobject` **/setcursor** –
The arguments to this method are either a NeWS cursor object or the parameters to identify one in a special dictionary (**/Cursors**). These are the image and the mask identifiers, respectively. They can be either a keyword such as **ptr** or an integer. A null second argument means that the mask is the next item in the

dictionary following the image specified in the argument.

/seteventmgr

`eventmgr|null` **/seteventmgr** –
Make the argument the event manager for this canvas. If the argument is **null**, there will be no event manager. Normally an application does not call this method to set its event manager, using **/activate** to do it implicitly.

/sethelp

`proc|null` **/sethelp** –
Install or remove the canvas's help procedure.

/setkeyconsumer

`bool` **/setkeyconsumer** –
This method stores its value as a variable to indicate whether the canvas will respond to input focus. True means interested in input focus. Simple applications using /ClassKeysInterest do not need to call this method.

/setlastfocustime

`time` **/setlastfocustime** –
Promotes a variable to store the time when the input focus was last in this canvas. Simple applications do not need to use this method.

/setmenu

`menu|null` **/setmenu** –
Install or remove a popup menu for this canvas. The menu is activated by the user pressing the MENU button when the pointer is over the canvas.

/setpaintproc

`proc` **/setpaintproc** –
This method redefines the painting procedure of a canvas (**/PaintCanvas**).

/settextfont

`font` **/settextfont** –
Set the text font object for a subclass or an instance. Note that because the font itself is stored, it may not agree with the font parameters specified by **/settextparams**. If there is a difference, the font takes precedence over the parameters.

`/settextparams`

`family pointsize encoding` **/settextparams**
Stores the parameters defining the text font for the canvas.

`/settransparent`

`bool` **/settransparent** –
Set the transparency for the canvas (the **/Transparent** key). If **false**, the canvas is made opaque and an interest in **/Damage** is automatically expressed

`/siblings+`

– **/siblings+** `array`
Return an array of NeWS canvases with the same parent as this canvas, including this canvas. The array is ordered by how the canvases (would) appear on the screen bottom first to top last.

`/siblings-`

– **/siblings-** `array`
Return an array of NeWS canvases with the same parent as this canvas, including this canvas. The array is ordered by how the canvases (would) appear on the screen top first to bottom last.

`/siblingsabove`

– **/siblingsabove** `array`
Return an array of NeWS canvases on top of this one in the canvas tree, from the canvas immediately above to the one on top.

`/siblingsbelow`

– **/siblingsbelow** `array`
Return an array of NeWS canvases below this one in the canvas tree, from the canvas immediately below to the one on the bottom.

`/size`

– **/size** `w h`
Return the width and height of the canvas in CTM.

`/stretchcorner`

`x y` **/stretchcorner** –
`event` **/stretchcorner** –
Reshape the canvas from user interaction with the farthest corner fixed, subject to minimum size constraints. If an event is supplied, its

position is used to determine where the user began the interaction by pressing a button. Otherwise, the coordinates are supplied directly.

`/textfont`

– **/textfont** `font`
Return the text font of the canvas.

`/textparams`

– **/textparams** `family pointsize encoding`
Return the text font parameters of the canvas.

`/tobottom`

– **/tobottom** –
Move the canvas to the bottom of its sibling list using the **canvastobottom** NeWS operator.

`/totop`

– **/totop** –
Move the canvas to the bottom of its sibling list using the **canvastotop** NeWS operator.

`/transparent?`

– **/transparent?** `bool`
Return the transparency of the canvas, **true** if transparent, **false** if opaque.

`/unmap`

– **/unmap** –
Unmap the canvas.

`/valid?`

– **/valid?** `bool`
Return **true** if the canvas is marked valid.

`/?validate`

– **/?validate** –
Validate the canvas if it is currently invalid.

`/validate`

– **/validate** –
Mark the canvas as valid. ClassCanvas does not itself interpret what validity means, but leaves it to subclasses to do so by overriding the indicated methods. For example, to a bag valid means not requiring layout at the moment.

Subclass Methods

/BuildCanvasSend

name | array /BuildCanvasSend proc
This method builds a callback for use with an interest in an event on this canvas. When a suitable event occurs and matches the interest the callback will send a message to this canvas. That message is the name of a method or an array (executable or not) that was given as the argument to **BuildCanvasSend**.
The callback is constructed so that it can identify this canvas from information in the event: the **/Interest** key of the event contains the interest, whose **/Canvas** key contains this canvas. The callback finds this canvas and sends the method name or array to it, making the array executable if necessary.

For example:

```
MenuButton /MenuNotify BuildCanvasSend
/DownTransition Canvas eventmgrinterest

PointButton {pop /OpenFrame} BuildCanvasSend
/DownTransition Canvas eventmgrinterest
```

application programmers will usually invoke **/BuildCanvasSend** indirectly through the simpler **/MakeInterest** method.

/Canvas

– /Canvas canvas
Return the NeWS canvas associated with this instance.

/CreateEventMgr

– /CreateEventMgr emgr
Create an event manager for this canvas. This method is automatically called by **/activate** if the canvas did not inherit an event manager from its parent.

/DamageInterest

– /DamageInterest interest
Return the canvas's stored damage interest.

/DescendantsRecurse

`can` **/DescendantsRecurse** `can can0 can1 can2 ...`
Return an array of canvases, the descendants (if any) of this canvas. The array is ordered by proximity to this canvas, children before grandchildren. Within each generation, top canvases come before bottom canvases. This method is used by **/descendants.**

/DisabledColor

`-` **/DisabledColor** `color`
Return the color used if the canvas is "disabled", the interpretation of which is left to subclasses. This method is a convenience to subclasses such as controls that have enabled and disabled states.

/EventMgr

`-` **/EventMgr** `eventmgr`
Return the event manager for this canvas.

/FillCanvas

`color` **/FillCanvas** `-`
Fill the entire canvas using **color.**

/FillCanvasInterior

`color inset` **/FillCanvasInterior** `-`
Fill the inside of the canvas using *color*. Leave an unpainted edge *inset* wide in canvas coordinates.

/FillColor

`-` **/FillColor** `color`
Return the fill color of the canvas.

/FilterNonInstances

`array` **/FilterNonInstances** `array'`
This method removes NeWS canvases that are not tNt canvases, ("non-instances") from arrays returned by methods that list canvases such as **/children+, /children-, /siblings+, /siblings-, /siblingsabove, /siblingsbelow, /descendants, /parents.**

/FixCanvas

`-` **/FixCanvas** `-`
This is the method that handles the repainting of damaged portions of a canvas. It should be overridden in those subclasses that are able to efficiently repaint a damaged area rather than the entire canvas.

`/GetCursorEncoding`

/GetCursorEncoding

`/HandleFix`

`event` **/HandleFix** –
This is the callback for the damage interest (**/DamageInterest**).

`/HandleHelp`

`event` **/HandleHelp** –
This is the callback for the help interest (**/HelpInterest**).

`/HandleMenu`

`event` **/HandleMenu** –
This is the callback for the menu interest (**/MenuInterest**).

`/HelpInterest`

– **/HelpInterest** `interest`
Return the help interest object.

`/MakeInterest`

`name callback action canvas` **/MakeInterest** `interest`
This method is used when overriding **/MakeInterests** to create a single interest that includes the *callback* argument as an executable match on the **/Action** field. The *name, canvas* and *action* arguments are the values placed in the **/Name, /Canvas** and **/Action** keys of the interest, respectively. The *canvas* argument is usually this canvas (**/Canvas**). When an event matches the interest, the callback is executed with the event on the stack.

`/MakeInterests`

– **/MakeInterests** `interestlist`
This method returns an array of the interests of this canvas. It is overridden to express additional interests in events over this canvas and to specify callbacks to execute when those events occur. As an example, consider:

```
/MakeInterests {
  /MakeInterests super send
     PointButton /my-method /DownTransition Canvas
     /MakeInterest self send
} def
```

This procedure sends **/MakeInterests** to its superclass to get any interests expressed there. The procedure then adds an interest in down transitions of the POINT button over this canvas, using the **/MakeInterest** method.

`/Mapped`

- **/Mapped** `bool`
Returns **true** if the canvas is mapped; the default is the same value as **/Transparent**.

`/MenuInterest`

- **/MenuInterest** `interest`
Returns the menu interest of the canvas.

`/MgrOwner?`

- **/MgrOwner?** `bool`
Returns **true** if this canvas created its own event manager.

`/PaintCanvas`

- **/PaintCanvas** -
Paint the entire canvas. Subclassers will want to override this method to correctly paint the canvas.

`/PaintScrolledArea`

`dx dy` **/PaintScrolledArea** -
This method is called by /scroll to paint the area of the canvas that was scrolled.

`/SetActive`

`bool` **/SetActive** -

`/SetGlobalCursor`

`/cursor /cursormask |`
`null` **/SetGlobalCursor** -

`/SharedDamage?`

- **/SharedDamage?** `bool`
Indicates that an interest in damage should be expressed if the canvas is opaque.

`/SharedHelp?`

- **/SharedHelp?** `bool`
Indicates that an interest in the help key should be expressed if there is a help procedure.

`/SharedMenu?`

- **/SharedMenu?** `bool`
Indicates that an interest in the MENU button should be expressed if there is a menu associated with the canvas.

/Siblings+

`can0|null` **/Siblings+** `can0 can1 can2... |`
Return the NeWS canvases with the same parent as this canvas, including this canvas. The array is ordered by how the canvases would be layered on the screen bottom first to top last. This method is used by **siblings+**.

/Siblings-

`can0|null` **/Siblings-** `can0 can1 can2... |`
Return the NeWS canvases with the same parent as this canvas, including this canvas. The array is ordered by how the canvases would be layered on the screen top first to bottom last. This method is used by **siblings-**.

/StrokeAndFillCanvas

`edgecolor inset`
`fillcolor` **/StrokeAndFillCanvas** –
Draw a border which is *inset* wide around the canvas using *edgecolor,* and fill the interior with *fillcolor.*

/StrokeCanvas

`color inset` **/StrokeCanvas** –
Draw a border *inset* units inside the canvas (in CTM) stroke it with *color.*

/StrokeColor

– **/StrokeColor** `color`
Return the stroke color.

/TextColor

– **/TextColor** `color`
Return the text color.

/TextEncoding

– **/TextEncoding** `array`
Returns the canvas's PostScript language encoding vector, an array containing the mapping of character names to character codes

/TextFamily

– **/TextFamily** `name`

/TextFont

– **/TextFont** `font`

/TextSize

– **/TextSize** `integer`
Returns the current point size of text.

/Transform

`x y w h` **/Transform** `x' y' w' h'`
This method is overridden to change the default CTM of the canvas. The following example

shows **/Transform** overridden to map the coordinates of a canvas onto the unit square..nf

```
/Transform { % x y w h => x' y' w' h'
  4 2 roll translate scale 0 0 1 1
} def
```

`/X0Y0FromUser`

– **/X0Y0FromUser** **x y**
Block until the user clicks the mouse on this canvas; return the location of the mouse click.

Class Variables

`/Active?`
This variable is **true** if the canvas is ready for user interaction.

`/AutoTargetMenu`
This variable is **true** if the canvas automatically sets itself as the target object of its installed menu when the menu is shown.

`/BorderStroke`
This variable is the size of the border around the canvas when painted by **/PaintCanvas.**

`/CanvasMenu`
This variable stores the canvas's menu.

`/CursorFont`
This variable stores the default cursor font of the canvas.

`/CursorImage`
This variable stores the default cursor image of the canvas.

`/CursorMask`
This variable stores the default cursor mask of the canvas.

`/Cursors`
This variable stores a dictionary containing the cursor objects of the canvas.

`/DragFrame?`
This variable indicates whether to drag the canvas (**false**) or just a wire frame (**true**) when moving the canvas.

`/EventMgrSet?`
This variable is **true** if the canvas has an event manager.

`/EventsConsumed`	This variable stores a list of all the events the canvas consumes.
`/HelpProc`	This variable stores the canvas's help procedure, if it has one.
`/KeyConsumer?`	This variable is **true** if the canvas is interested in keyboard input, regardless of whether the canvas actually consumes them or passes them on.
`/LastFocusTime`	This variable stores the last time the canvas had the input focus.
`/Retained`	This variable is **true** if the canvas is retained.
`/SaveBehind`	This variable is the NeWS /SaveBehind hint to the server about how to handle damage to underlying canvases when this canvas is put on screen.
`/StdCursorFont`	This variable stores a dictionary of cursor fonts.
`/Transparent`	This variable is **true** if the canvas is transparent.

see also:

Object

ClassCommandFrame

Subclass of **ClassFrame**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

This class is not for direct use; rather it supports frame (window) subclasses, such as the OPEN LOOK window frames. If you need a simple window frame for immediate use, consider one of the OPEN LOOK frames:

- OpenLookBaseFrame
- OpenLookPropertyFrame

- OpenLookNoticeFrame
- OpenLookCommandFrame
- OpenLookHelpFrame

see also:

ClassFrame

ClassContainer

Subclass of **ClassBag**
Source file: **bag.ps**

This class can be directly instantiated, but is usually subclassed.

A container is a bag with defined border areas and a special client, named /Client. When the bag is reshaped, the bag takes for itself a fixed amount of space for the borders, specified by class variables. All remaining space goes to /Client.

Examples of subclasses of ClassContainer are frames and panes.

Direct Methods

`/client`

– **/client** `client|null`
Return the container's designated main client, the **/Client** object, or **null** if empty.

`/destroy`

– **/destroy** –
Destroy the container and its client. Refer to */destroy* in *ClassBag* for more information on this method.

`/destroydependent`

– **/destroydependent** –
Override this method for any client that should not be destroyed at the same time as the container and other clients.

`/fitclient`

`w h` **/fitclient** `w' h'`
Add the container's borders to the size of **/Client** to obtain the size the container must be to fit a client of the given size.

/location

- **/location** `x y`
Return the location of the bag's origin in CTM coordinates.

/minsize

/minsize `minwidth minheight`
The minimum size for a container is by default the minimum size of /Client with the border sizes added.

/move

`x y` **/move** -
Move the origin of the container to the point (x,y) in the coordinates of the CTM.

/new

`client|null parentcanvas` **/new** `instance`
`clientclass parentcanvas` **/new** `instance`
`[args clientclass] parentcanvas` **/new** `instance`
This method creates a new container, calling **/newinit** to perform initialization once the instance is created. The arguments to **/new** are:

- the parent canvas of the container being created
- the client canvas of the container, or the class from which to create an instance to be the client, or **null**. The client instance or the class to be instantiated forms See **/addclient** in *ClassBag* for more details on specifiying the client instance or a class to be instantiated.

/newinit

`client|null` **/newinit** -
This method is called from **/new** and consumes the *client* argument while initializing the new client.

/preferredsize

- **/preferredsize** `preferredwidth preferredheight`
Calculate the preferred size of the container, by default, the preferred size of its designated client plus the size of border areas.

/reshape

x y w h /reshape –
Reshape the container to the given size. Later, during layout the borders (and any secondary clients) are given their fixed amount of space, and the designated client absorbs all remaining space.

/setclient

newclient|null /setclient oldclient|null
Set **/Client** for the container; return the previous **/Client** or **null** if there was none. Supplying **null** removes the existing client, if there was one, without setting a new one.

/size

– /size w h
Return the current size of the container in the coordinates of the CTM.

/unfitclient

w h /unfitclient w' h'
Subtract the borders from the size of the container to obtain the size **/Client** must be to fit into the container.

Subclass Methods

/BorderHeights

– /BorderHeights number
This method returns the total border height of the container, the sum of the heights of the header at the top plus the footer at the bottom. Override this method if your container has a different layout or geometry.

/BorderWidths

– /BorderWidths number
This method returns the total border width of the container, the sum of the widths of the left and right borders. Override this method if your container has a different layout or geometry.

/LayoutClient

– /LayoutClient –
Position the client in the container as specified by the **Border** variables:

- **/BorderLeft** units from the left edge,
- **/BorderRight** units from the right edge,
- **/BorderBottom** units from the bottom, and
- **/BorderTop** units from the top.

  Override this method for containers with a different arrangement or geometry.

Class Variables

`/BorderBottom` This variable should be overridden to define how far from the bottom of the container the client is placed.

`/BorderLeft` This variable should be overridden to define how far from the left edge of the container the client is placed.

`/BorderRight` This variable should be overridden to define how far from the right edge of the container the client is placed.

`/BorderTop` This variable should be overridden to define how far from the top of the container the client is placed.

ClassControl

Subclass of **ClassCanvas,ClassTarget**
Source file: **control.ps**

This class should be subclassed rather than directly instantiated.

A control is a canvas that responds to user input with visual feedback. A control consists of:

- a value and some way to dynamically display it on screen.
  The value can be any PostScript language object, although subclasses usually restrict the value.
- a notification procedure, or callback,
  which is called whenever the control's value changes as the result of user interaction. The notification procedure is executed with the control object on the operand stack, and can therefore find any specific information about the control (for example, its value);
- a target object
  to which the notification procedure can send a message. Often, user action on a control will do more than simply change the control's value. The target allows the control to specify another object to notify with its callback procedure. For example, if you want a menu to appear in response to pressing a mouse button over a control, you would make the menu the target for the control's callback.
- a tracking process,
  which is a simple event manager that watches for activity on a mouse button identified in the class variable /ControlButton. (Usually the SELECT button is designated.) The tracking process is initiated when the button is pressed and watches for /MouseDragged, /EnterEvent, /ExitEvent and /UpTransition events. When the /UpTransition occurs the notification procedure is executed.
- a state of enabled or disabled:
  when the control is disabled, it no longer responds to user input. Usually, the functionality that displays the control's value will indicate the control's state as well

The simplest example of a directly usable control is OpenLookButton. In this case the control's value is represented with a graphic (an instance of a subclass of ClassGraphic). The graphic is drawn (and redrawn) to represent the value of the graphic. having a different appearance when enabled than when disabled.

Direct Methods

`/activate`

– **/activate** –
Activate event management for this control; express the interests returned by the **/makeinterests** method. Create an event manager for the control if one cannot be found in one of its parent canvases. An application does not usually call this method directly. Instead, controls are commonly grouped together in a bag, and the bag controls activation for all of its clients.

`/active?`

– **/active?** `bool`
Return **true** if event management has been activated on the control.

`/callnotify`

– **/callnotify** –
Call the control's notify procedure unconditionally, with the control itself on the operand stack.

`/checknotify`

`object|null` **/checknotify** –
This method sends **/CallNotify?** to the control and if **true** is returned, the notify procedure is called. This mechanism provides conditional notification based on criteria specified by the subclasser in overriding **/CallNotify?**.

`/cleartarget`

`object|null` **/cleartarget** –
Selectively clear the target. If null is given, set the target to null. If an object is given, set the target to null only if the target is the object.

`/deactivate`

– **/deactivate** –
Turn off event management for this control. If the event manager is owned by this control, destroy it.

`/destroy`

– **/destroy** –
Turn off tracking if it is on, and destroy the control.

`/disable`

– **/disable** –
Set the control to the disabled state, in which user interaction is not allowed. Generally the

appearance of the control on screen changes whed disabled.

/enable

- **/enable** -
Set the control's state to enabled, and display it accordingly with **/PaintEnabledState**.

/enabled?

- **/enabled?** bool
Return **true** if the control is currently enabled, **false** if it is disabled. When disabled, the control does not respond to user input.

/location

- **/location** x y
Return the location of the origin of the control relative to the CTM.

/minsize

- **/minsize** w h
Return the smallest size the control can be and still appear intelligible.

/move

x y **/move** -
Move the origin of the control to the specified location in the coordinates of the CTM.

/new

{notifyproc}|null parentcanvas **/new** -
Create a new instance of the control, specifying the control's parent canvas and its notification procedure. When subclassing, additional arguments consumed by **/newinit** should appear before the notification procedure.

/newinit

{notifyproc}|null **/newinit** -
This method is called from **/new** to after creating an instance. **/newinit** does superclass initialization, sets the notify procedure to the supplied argument and initializes the variable that stores the control's last notify value.

/notifiedvalue

- **/notifiedvalue** any
Return the value the control had the last time the notification procedure was called.

/notifyproc

- **/notifyproc** proc
Return the control's current notify procedure.

`/preferredsize`

`-` **/preferredsize** `width height`
Return the control's preferred size. For most controls (except OPEN LOOK scrollbars), this will be the minimum size, **/minsize**.

`/reshape`

`x y w h` **/reshape** `-`
Reshape the control to fit the bounding box specified by the arguments, in the coordinates of the CTM. This method allows the toolkit or the application to position the control on its parent canvas.

`/sendtarget`

`args /method` **/sendtarget** `results`
Send *method* and its arguments to the current target object. This method is often used inside the control's notification procedure to send a message to the target object of the control.

`/setnotifyproc`

`proc` **/setnotifyproc** `-`
Set the control's notify procedure, overwriting the previous one.

`/settarget`

`object` **/settarget** `-`
Set the target object for the control. This should be done just after the control is created, before the notification can be invoked. Sending a message to a nonexistent target results in an error.

`/setvalue`

`any` **/setvalue** `-`
Set the value of the control. It is a subclass issue to define the set of legal values for a control, and what action to take given an illegal value.

`/size`

`-` **/size** `w h`
Return the width and height of the control in the coordinates of the CTM.

`/target`

`-` **/target** `object`
Return the target object of the control.

`/trackinterests`

`-` **/trackinterests** `[interests]`
Return an array containing the tracking interests

of the control. This method is not usually called directly by an application.

`/trackmgr`

– **/trackmgr** `eventmgr|null`
Return the tracking event manager for this control.

`/trackoff`

– **/trackoff** –
Turn tracking off for the control. Destroy the tracking event manager as soon as it has handled any pending events.

This method is not usually called directly by an application, but is invoked automatically through **/StopTracking** when the user releases **/ControlButton** after pressing it over the control.

`/trackon`

– **/trackon** –
Turn on tracking in response to the user pressing the indicated mouse button (**/ControlButton**) over the control. If there is no tracking event manager, create one and call /MakeTrackInterests to express the control's tracking interests.

This method is not usually called directly by an application, but is invoked automatically through **/StartTracking** when the user presses down **/ControlButton** over the control.

`/value`

– **/value** any
Return the current value of the control. The set of values will be restricted by most subclasses, which have the responsibility for ensuring that illegal values do not get stored.

Subclass Methods

/BuildTrackInterest

`name action canvas`
`method` **/BuildTrackInterest** `interest | -`
This method is called by **/MakeTrackInterests** to build one interest from the arguments supplied. The *name, action,* and *canvas* arguments go into the interest as their respective key values. *method* is the name of the callback procedure; it is a method in the control class that handles the event

/CallNotify?

`object | null` **/CallNotify?** `bool`
Override this method to define the criteria for calling the notification procedure. The default criterion is that the current value of the control is different than the value when the last notification was performed (**/NotifiedValue**).

/ClientDown

`event` **/ClientDown** –
Override this method to implement special behavior at the start of tracking.

/ClientDrag

`event` **/ClientDrag** –
Override this method to provide a tracking interest callback for every mouse movement inside the control.

/ClientEnter

`event` **/ClientEnter** –
Override this method to provide a tracking interest callback for mouse movements into the control.

/ClientExit

`event` **/ClientExit** –
Override this method to provide a tracking interest callback for mouse movements out of the control.

/ClientRepeat

`event` **/ClientRepeat** –
a periodic action controlled by the **/ClientRepeatTime** and **/ClientStartTime** variables.

`/ClientUp`

`event` **/ClientUp** –
Override this method to implement special behavior at the end of tracking.

`/EndTracking`

`event` **/EndTracking** –
This method is called automatically in response to a tracking interest in the **/UpTransition** of the **/ControlButton.** It calls **/ClientUp** to do any special handling (none by default) and then turns off tracking with **/trackoff.**

`/EventHandler`

`event` **/EventHandler** –
When the control is initialized, it expresses an interest in **/ControlAction** on the **/ControlButton,** with this method as the callback. By default, the interest is in the **/DownTransition** of the **PointButton.** When an event occurs that matches the interest, this method is called with the event on the operand stack.
Override this method to define the control's behavior when that event occurs. Typically, the override will include a call to**/StartTracking.**

`/EventToXY`

`event` **/EventToXY** `x y`
Extract the XLocation and YLocation from the event, and return them.

`/MakeTrackInterests`

– **/MakeTrackInterests** `[interests]`
Override this method to specify which tracking interests a subclass needs. This method is called from /trackinterests to generate an array of interests

`/PaintEnabledState`

`bool` **/PaintEnabledState** –
Override this method if the control's appearance on screen depends on whether it is enabled or disabled. This method is called by **/enable** and **/disable:**

- with the control as the current canvas

- inside **gsave ... grestore**
- with the boolean arguments **true** and **false** respectively.

`/PaintValue`

`newvalue` **/PaintValue** –
This method is SubClassResponsibility and must be overridden to display the value of the control.

Subclassers should note that this method is called by **/setvalue** with the control's new value on the stack, while the previous value is still available through **/value**. Thus, old and new values of the control are both available to this method if it needs them, for example, to perform incremental painting.

`/StartTracking`

`event` **/StartTracking** –
Start event tracking for this control, if the control is enabled. This method is called in response to a tracking interest in the **/DownTransition** of the **ControlButton**. It calls **/ClientDown** to do any special handling (none by default) and then turns on tracking with **/trackon**.

Class Variables

`/Active?`

This variable is **true** if the control has event management turned on.

`/ClientRepeatTime`

This variable is the interval for a timer interest controlling when the **/ClientRepeat** method is called.

`/ClientStartTime`

This variable is the starting time for a timer interest controlling when the **/ClientRepeat** method is called.

`/ControlAction`

This variable specifies the action of the **/ControlButton** that starts tracking. By default it is the **/DownTransition**.

`/ControlButton` This variable specifies which mouse button the control is interested in, by default it is the **PointButton.**

see also:

ClassCanvas,ClassTarget

ClassDialControl

Subclass of **ClassControl**
Source File: **dial.ps**

This class should be subclassed rather than instantiated.

ClassDialControls is the basis for "analog" controls, which have a bounded numerical value that changes in resonse to user interaction. Commonly used dial controls are scrollbars and sliders.

see also:

ClassControl, OpenLookVerticalScrollbar

ClassFrame

Subclass of **ClassContainer**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

ClassFrame is designed for the management of windows (frames) on the frame-buffer. It is an instrinsic class supporting the OpenLook frames:

- OpenLookBaseFrame,
- OpenLookCommandFrame,
- OpenLookHelpFrame
- OpenLookIconFrame
- OpenLookNoticeFrame
- OpenLookPropertyFrame

see also:

ClassContainer, OpenLookBaseFrame

ClassHelpFrame

Subclass of **ClassFrame**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

This class is not for direct use; rather it supports subclasses of window frame, such as the OPEN LOOK window frames. If you need a simple window frame for immediate use, consider one of the OPEN LOOK frames: OpenLook-BaseFrame, OpenLookPropertyFrame, OpenLookNoticeFrame, OpenLookCommandFrame, OpenLookHelpFrame.

ClassIconFrame

Subclass of **ClassFrame**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

This class is not for direct use; rather it supports frame (window) subclasses, such as the OPEN LOOK window frames. If you need a simple window frame for immediate use, consider one of the OPEN LOOK frames: OpenLook-BaseFrame, OpenLookPropertyFrame, OpenLookNoticeFrame, OpenLookCommandFrame, OpenLookHelpFrame.

ClassMenu

Subclass of **ClassSelectionList**
Source file: **menu.ps**

This class should be subclassed rather than directly instantiated.

This class is suports hierarchical, pop-up, pinnable menus. It is not immediately instantiatable, but forms the underpinning for OpenLookMenu.

ClassPropertyFrame

Subclass of **ClassFrame**
Source file: **frame.ps**

This class should be subclassed rather than directly instantiated.

This class is not for direct use; rather it supports frame (window) subclasses, such as the OPEN LOOK window frames. If you need a simple window frame for immediate use, consider one of the OPEN LOOK frames: OpenLook-BaseFrame, OpenLookPropertyFrame, OpenLookNoticeFrame, OpenLookCommandFrame, OpenLookHelpFrame.

ClassSelectionList

Subclass of **ClassCanvas,ClassTarget**
Source file: **selectlst.ps**

This class should be subclassed rather than directly instantiated.

In a selection list, a single canvas manages a grid of regularly spaced items which can be independently selected via the mouse. This class is the basis for menus (OpenLookMenu) and setting controls (Exclusive, NonExclusive, and Choggles).

ClassTarget

Subclass of **Object**
Source file: **target.ps**

This class can be directly instantiated.

ClassTarget, typically used as a mix-in class, provides facilities to any class whose instances need to send messages to other objects. Controls and menus are examples of such classes, as callbacks from these classes typically involve sending a message to some other object. The target connection is one-way and is maintained as long as the targeted object persists.

The connection is actually implemented as a soft reference to the targeted object. If the targeted object should become obsolete the soft reference is removed and the target is set to null.

Direct Methods

`/cleartarget`

`object|null` **/cleartarget** –
Selectively clear the target. If null is given, the target is cleared. If *object* is given, the target is cleared only if the *object* and the traget are the same. This ensures that the target cannot be incorrectly cleared.

`/destroy`

– **/destroy** –
Clear the target (with **/cleartarget**) and then proceed with the normal object destruction.

`/sendtarget`

`args /method` **/sendtarget** `results`
Send the method and arguments to the targeted object. Any menu or control can have a "target" which you can set, get, and send to. For example, if you have a button that needs to send a message to object *foo* when the button's callback is executed:

```
/b (Hello) {/foo-method /sendtarget 3 -1 roll send}
framebuffer /new OpenLookButton send def
...
foo /settarget b send
```

When the button is pressed, *foo-method* will be sent to *foo.*

You can change the target at any time if you want to make your control send to different objects at different times. An error results from sending anything to a null target.

`/settarget`

`object` **/settarget** –
Set the target object, overwriting the previous target, if there was one.

`/target` — `- /target object`
Return the target object, which has a soft reference to it.

Subclass Methods

`/ChangedTarget` — `old-target /ChangedTarget -`
This method is called whenever a target changes. Override it to add special processing if necessary.

`/FreshTarget` — `- /FreshTarget -`
This method is invoked whenever the target is changed from null to an object. Override this method to provide special processing for this case.

`/ObsoleteTarget` — `/ObsoleteTarget -`
This method is called when the target becomes obsolete. Override it to add special processing.

`/RemovedTarget` — `/RemovedTarget -`
This method is called whenever a target is removed (becomes **null**). Override this method to provide special processing for this case.

see also:
OpenLookMenu, ClassControl

ClassTextControl

Subclass of **ClassControl**
Source file: **txtctrl.ps**

This class should be subclassed rather than directly instantiated.

A text control accepts user keystrokes, displays them and calls the supplied notify proc. It interacts with the UI-independent selection mechanism, providing simple text selections.

Direct Methods

`/callnotify`

– **/callnotify** –
Call the client's notifyproc, and also make a copy of the string. The copy will not be affected by subsequent user keystrokes, so **/checknotify** will be able to determine whether the value has changed.

`/checknotify`

`object|null` **/checknotify** –
Call the client's notifyproc only if the current text differs from the text as of the last call to **/callnotify.** This method is called automatically when the text control loses the input focus, or when the user types the RETURN key.

`/cleartarget`

`object|null` **/cleartarget** –
Selectively clear the target used by the **/sendtarget** method. This method is called automatically when the text control is destroyed. See ClassTarget for more about targets and their uses.

`/delchar`

`n` **/delchar** –
Delete **n** characters from the text, starting at the current caret position. If **n** is negative, delete characters to the left of the caret, otherwise to the right.

`/delspan`

`left rightplusone` **/delspan** –
Delete a span of characters from the text, independent of where the caret is. If the caret and/or a selection is within the deleted region, it is adjusted accordingly. Character positions start with zero; thus **left** is the number of characters before the first to be deleted, and the number of characters deleted is **rightplusone** minus **left.**

`/delword`

– **/delword** –
Delete characters backward from the caret position until (a) at least one alphanumeric has been deleted and (b) the character ahead of the caret is not alphanumeric. The method

/AlphaNumeric? is used to determine whether a character can be part of a word.

`/demo`

– **/demo** `instance`
Create a sample text control whose parent is the framebuffer. The control is actually an instance of the default subclass (i.e., OpenLookTextControl).

`/destroy`

– **/destroy** –
Destroy this control. In addition to clearing the target and removing the canvas from the canvas tree, etc., this method disables the caret, in case for example the caret needs to stop a forked process used to implement blinking. (The default ClassCaret does not.)

`/disable`

– **/disable** –
Cause the control to stop responding to user actions. The control repaints to display its disabled status, and will no longer accept the input focus.

`/enable`

– **/enable** –
Cause the control to respond to user actions, provided that it is not a read-only text control. The control repaints to display its enabled status, and will become the input focus if the user clicks the mouse over it.

`/enabled?`

– **/enabled?** `bool`
Return **true** if the control is currently enabled, even if it is read-only.

`/fitcaret`

– **/fitcaret** –
Call **/FitCaret**, which does nothing in this class. Subclassers may override **/FitCaret** to provide a method that brings the caret into the visible region of the control.

`/fontoffset`

– **/fontoffset** `int`
Compute the vertical offset for drawing characters without cutting them off at the bottom of

the canvas, and also leaving enough room for the caret. The value is 'promoted', i.e., the first time **/fontoffset** is called it stores the resulting value in the instance under the name **/fontoffset**, so that subsequent calls are much faster. The cached value is discarded (unpromoted) if the canvas becomes invalid, e.g. if **/settextparams** is called.

`/inserttext`

`char` **/inserttext** –
`string` **/inserttext** –
Insert a character or a string into the text at the current caret position. The caret is moved to the end of the inserted text. If a pending-delete primary selection spans the caret, the selected text is deleted before adding the new text. Any other existing selections are adjusted appropriately.

`/invisiblecaret`

– **/invisiblecaret** –
Make the caret invisible, usually in preparation for other operations that will cause painting. This method is not normally used at all, since the methods that cause painting (such as **/inserttext**, **/delchar**, etc.) all call **/InvisibleCaret**, which assumes that the text control is the current canvas.

`/location`

– **/location** `x y`
Return the location of the origin (lower left corner) of the text control relative to the CTM.

`/minsize`

– **/minsize** `minwidth minheight`
Return the minimum size for this control. The minimum height is determined by the font size, and the width is determined by the font and by the number of characters that the control is supposed to leave room for, as set using **/setdisplaychars**.

`/move`

`x y` **/move** –
Move the origin (lower left) of the text control to

the specified location in the coordinates of the CTM.

`/new`

callback parent /new instance
Return a new text control instance with the specified callback procedure
and parent canvas. The control initially contains no text, but does contain a caret.

`/newinit`

callback /newinit –
This method does **/newinit super send** to store the callback procedure, then performs other initialisation specific to text controls.

`/notifiedvalue`

– /notifiedvalue string
Return the value the control had the last time **/callnotify** was sent.

`/notifyproc`

– /notifyproc proc
Return the current callback proc without invoking it.

`/painttext`

n /painttext –
Paint the text, starting with the character at position **n**. If character **n** is off the left edge of the canvas, painting starts at the edge of the canvas. If **n** < 0, all the text is painted and the remainder of the canvas is cleared; otherwise the canvas is left unchanged beyond the end of the text. This method is intended mainly for use from other methods; clients should generally send **/paint** to paint the control. **/painttext** operates by setting the current canvas and font, then calling the internal methods **/InvisibleCaret**, **/PaintText**, and **/VisibleCaret**.

`/preferredsize`

– /preferredsize width height
Text controls do not presume any preferred size beyond the minimum required to display a certain number of characters. Thus they use the default **/preferredsize** method, which simply calls **/minsize**.

/removefocus

`event` **/removefocus** –
Handle the input focus being moved to another canvas. This method disables the caret and calls **/checknotify,** which calls the client's notifyproc if the text has been changed.

/reshape

`x y w h` **/reshape** –
Reshape the control to fit the bounding box specified by the arguments, in the coordinates of the CTM.

/restorefocus

– **/restorefocus** –
Activate the caret if the control is enabled. This method is called automatically when the text control is given the input focus.

/scroll

n **/scroll** –
Scroll the text to make a different portion visible within the limits of canvas. The left edge of the canvas always marks the beginning of a character; scrolling changes the character at this location. A positive number nn scrolls to the right; a negative number scrolls to the left.

/sendtarget

`args /method` **/sendtarget** `results`
This method can be invoked within the notifyproc in order to send a message to some other instance. Many clients will not need to use this indirection. See ClassTarget for more details.

/setcolors

`strokecolor fillcolor`
`textcolor` **/setcolors** –
Set the colors for painting the text and its background (fillcolor). Null as an argument means do not change that value. The control swaps the fillcolor and textcolor to highlight selections. Subclasses may use the strokecolor for underlining or similar ornamentation, though OpenLook-TextControl uses the textcolor so that it can be included in the highlighting.

/setdisplaychars

n **/setdisplaychars** –
Set the minimum number of characters that the

control should be able to display. This number is used in computing **/minsize** for the control. The number can be retrieved via the **/DisplayChars** variable.

/setnotifyproc

`proc` **/setnotifyproc** –
Store the proc as the new callback to be used by **/callnotify**, in place of the one provided to **/new**.

/setposition

`event` **/setposition** –
`n` **/setposition** –
Move the caret to the specified position. Position 0 puts the caret to the left of the first character. If an event is given, the coordinates in the event are resolved to the nearest character boundary. The current caret location can be obtained from the variable **/Left**.

/setreadonly

`bool` **/setreadonly** –
Make the text read-only or not, depending on the bool. If the text becomes read-only, the canvas is removed from the list of potential input foci; if it becomes writeable it is added back. A read-only text control behaves much the same as a disabled one (see **/disable**), the main difference being that a disabled control will typically be painted light gray, whereas a read-only one will paint normally.

/settarget

`object` **/settarget** –
Set the target used by the **/sendtarget** method. See ClassTarget for more about targets and their uses.

/settextparams

`family pointsize encoding` **/settextparams**
Set the text parameters that determine the font for the control. If any of the arguments is null, that parameter is not changed. The control then marks itself as invalid (i.e., calls **/invalidate**) so that various cached values such as **/fontoffset** and the font itself will be recomputed.

`/setvalue`
: **string /setvalue -**
Replace the contents of the text control with the given string. If the string differs from the old contents, the control is repainted.

`/size`
: **- /size w h**
Return the width and height of the control in the CTM.

`/starttext`
: **- /starttext -**
This method activates the caret. It is called from **/restorefocus.**

`/stoptext`
: **- /stoptext -**
This method disables the caret. It is called from **/removefocus** and **/destroy.** Clients may also need to send **/stoptext** to newly created text controls to disable their initial carets.

`/strlen`
: **- /strlen n**
Return the number of characters of text currently stored in the control. This method is equivalent to, but more efficient than, **/value** *length.*

`/target`
: **- /target object**
Return the current target (if any) used by **/sendtarget.** See ClassTarget for more details.

`/value`
: **- /value string**
Return the contents of the control as a PostScript string. If the string is empty, a zero-length string is returned, *not* null.

`/visiblecaret`
: **- /visiblecaret -**
Make the caret visible, usually after other painting has finished. This method is not normally used at all, since the methods that cause painting (such as **/inserttext, /delchar,** etc.) all call **/VisibleCaret,** which assumes that the text control is the current canvas.

Subclass Methods

/AlphaNumeric?

n **/AlphaNumeric?** bool
Determine whether the character at the specified index is part of a word. This method is called by **/delword,** and also when making word-level selections. The default method texts for letters, digits, and underscore. See **/AlphaNumericT-able.**

/CaretPosition

– **/CaretPosition** x
Return the x-coordinate of the caret relative to the left edge of the canvas.

/DeHighlight

start end **/DeHighlight** last
Paint the characters in positions *start*start through *end*-1, and return the index of the next character to be painted. The characters are painted using the canvas's **/FillColor** as back-ground and **/TextColor** for the text and under-line. This method is called by **/PaintText** and in turn calls **/PaintNText.**

/EOL

– **/EOL** –
This method is called when either a RETURN (^M) or NEWLINE (^J) character is typed into the control. It calls **/checknotify.**

/FitCaret

/method **/FitCaret** bool
Scroll the text if desired so as to bring the current caret position into the region between the left and right edges of the canvas. Return **true** if any scrolling was done, so the caller will know to repaint the text. (**/FitCaret** does not itself do any painting.) The parameter is the name of the operation just performed, one of: **/delspan, /fitcaret, /inserttext,** or **/setposition.** The default **/FitCaret** never does any scrolling. Subclassers may override this behavior to cause scrolling after selected operations.

/Highlight

start end **/Highlight** last
This is the same as **/DeHighlight** (q.v.) except

the background and foreground colors are reversed. It is called by **/PaintText** for painting characters within the primary selection.

/InterestingRank

`rank` **/InterestingRank** `bool`
Return **true** if the selection rank is one that the text control knows how to deal with. By default the only interesting ranks are **/PrimarySelection** and **/SecondarySelection.** (**/ShelfSelection** is handled by the global UI mechanism.) For each interesting rank there is also a correspondingly named instance variable, used for maintaining information about that selection. (These variables are of no interest to clients or subclassers, except insofar as subclassers must be careful not to use those variable names for other information.)

/InvisibleCaret

`-` **/InvisibleCaret** `-`
Send a message to the caret to make it vanish, usually in preparation for other painting. This method assumes that the text control is the current canvas. Many methods use the sequence: **/TextBegin, /InvisibleCaret,** paint some portion of the text, /VisibleCaret, **/TextEnd.**

/PaintText

`n` **/PaintText** `-`
This method performs most of the work of **/painttext** (q.v.). It can assume that the text control's canvas is the current canvas, and the control's font is the current font. Subclassers may override this method to provide additional painting; e.g., OpenLookTextControl repaints the scroll buttons after the text is drawn.

/PaintNText

`textcolor backcolor start`
`end` **/PaintNText** `last`
This method is called by **/Highlight** and **/DeHighlight** to paint portions of the text using the given colors for the text and background. The characters painted are those in positions

*start*start through *end*end-1. This range may be empty. **/PaintNText** returns the index of the next character to be painted, the larger of *end*end and *start*start. This method assumes that the text control is the current canvas and the control's font is the current font.

`/Scroll`

n /Scroll –
This is the same as /scroll except no painting is done. Subclassers may use /Scroll when overriding /FitCaret, to perform scrolling while leaving any painting for later.

`/TextBegin`

– /TextBegin –
Save the graphics context, then set the text control's canvas and font as the current canvas and font. This method is called at the beginning of several other methods to establish the context for painting and other CTM-based operations.

`/TextEnd`

– /TextEnd –
Restore the graphics context saved by **/TextBegin.**

`/VisibleCaret`

– /VisibleCaret –
Send a message to the caret telling it to become visible, and where it should be located. This method assumes that the text control is the current canvas.

Class Variables

`/AlphaNumericTable`

This is a dictionary whose keys are single characters (integers). The associated values are ignored. A character appears as a key if and only if that character is considered part of a word. This dictionary is used by **/AlphaNumeric?,** and can be overridden by subclassers to change the definition of word-selection and **/delword.** Note: Subclassers who wish to change this dictionary should be careful

to copy it first; otherwise the changes will affect all text controls. That is, do *not* subclass like this:

```
%add dollarsign and percent to alphanumeric set
AlphaNumericTable begin
($%) {null def} forall
end
```

Instead, write this:

```
% add dollarsign and percent to alphanumeric set
/AlphaNumericTable dictbegin
AlphaNumericTable {def} forall
($%) {null def} forall
dictend def
```

`/DisplayChars` — The minimum number of characters that the control should be able to display. The default value is 5, unless overridden by subclassing or by **/setdisplaychars.**

`/Left` — The number of characters to the left of the caret; hence, the current caret position.

`/ReadOnly?` — **True** if the control is read-only. This value defaults to **false**, but can be overridden by subclassers or **/setreadonly.**

see also:

ClassControl

FlexBag

Subclass of **ClassBag**
Source file: **bagutils.ps**

This class can be directly instantiated

A FlexBag is a general purpose bag whose clients are layed out using executable code passed in with them during /addclient. FlexBags are particularly suited to the task of relative positioning.

Direct Methods

/addclient

`name|null [client] /addclient -`
`name|null [compass_point position_proc client] /addclient -`

The naming and instantiation of a flexbag client are the same as for ClassBag. In particular, the *client* argument may be either an instance or a class to instantiate, with necessary arguments. For more details, see the **addclient** method in *ClassBag*.

If baggage is supplied, it consists of a reference point on the client (*compass_point*) and a positioning procedure (*position_proc*) that determines where to place that point on the client in the bag. The client's reference point is specified with a compass-style notation:

corners of the client: /nw, /sw, /ne, /se
midpoints of the client's edges: /n, /s, /e, /w
center of the client: /c

The positioning procedure is any executable that returns an (x,y) coordinate location. This position is where the reference point of the client is placed when the flexbag is layed out. The procedure can simply return two fixed values. Or, it can perform calculations of its own based on some intrinsic properties of the client. Or the position procedure can use the utility procedures supplied by class FlexBag to perform positioning relative to other clients of the flexbag. These procedures (/HEIGHT, /WIDTH, /POSITION, /XYADD, /XYSUB) are described below.

Note that both of the reference point and positioning procedure may be unspecified, in which case a default value will be used (see **/setlayoutspec**).

/baggage

`client` **/baggage** `[compass_point position_proc]`
Return the positioning data for the client, its reference point in compass notation and its positioning procedure.

/clientcount

– **/clientcount** `n`
Return the number of clients currently in the bag.

/clientlist

– **/clientlist** `[client1 client2 ...]`
Return an array of clients in the same order as they were inserted into the bag.

/demo

– **/demo** `frame_instance`
This method creates a sample flexbag that illustrates the use of absolute, relative, and default positioning. The frame object that encloses the bag is left on the operand stack by **/demo**. To use this method be sure to load the demo code with the **/IncludeDemos?** flag.

/destroy

– **/destroy** –
Destroy the bag and its clients. Refer to the **/destroy** method in *ClassBag* for the additional information on the use of this method.

/layoutspec

– **/layoutspec** `compass_point position_proc`
Return the default layout specification, which is used if no baggage was specified when adding a client to the bag. Initially, there is no default layout; it must be specified with **/setlayoutspec**.

/location

– **/location** `x y`
Return the location of the origin of the bag in the coordinates of the CTM.

/minsize

/minsize `minwidth minheight`
Compute the minimum acceptable size for this bag, based on the actual sizes of its clients and the padding between them. For a flexbag this requires a heuristic calculation. This problem can be circumvented by calling **/lockminsize**.

/move

`x y` **/move** –
Move the origin of the canvas to the specified location in the coordinates of the CTM.

/new

`parentcanvas` **/new** `instance`
Create a new flexbag.

/padding

– **/padding** `padding_width padding_height`
Return the sizes of the padding between clients used by **/minsize** to avoid unrelated clients being placed too close together.

/preferredsize

– **/preferredsize** `preferredwidth preferredheight`
Calculate the preferred or "ideal" size of the bag, which by default is its minimum size.

/removeclient

`client|name|n` **/removeclient** `oldclient true`
`client|name|n` **/removeclient** `false`
Remove the client given, named or indexed in the argument. The method returns **true** and the client object if the client is found, otherwise it returns **false.**

/reshape

`x y w h` **/reshape** –
Reshape the bag to the dimensions given and invalidate it. This results in the bag being layed out as the first step in painting it.

/sendclient

`<args> /method /name` **/sendclient** `results`
Send the given method with arguments to the named client. An error results if the client is not present in the bag.

/setbaggage

`client [compass_point position_proc]` **/setbaggage** –
Store the baggage for the bag. This consists of a compass point and a positioning procedure, as described under **/addclient**.

/setlayoutspec

`compass_point|null position_proc|null` **/setlayoutspec** –
Set the default positioning parameters for the

bag. This specification is used whenever a specification is not given when the client is added. If either argument is **null**, the corresponding parameter is not changed.

`/setpadding`
: `width height` **/setpadding** –
Set the padding between clients used in calculating the minumum size of the bag.

`/size`
: – **/size** `w h`
Return the width and height of the bag in the coordinates of the CTM.

Utility Methods

The following methods are defined in class FlexBag for use in positioning clients of a flexbag. Used in the position procedure (*position_proc*) of a client, they are executed by the **/layout** method of the bag.

The *client* argument given to these methods can be:

- a client of the flexbag,
- `Previous`, indicating the previous client (the one that was layed out immediately before this client
- `Current`, indicating this client,
- `self`, indicating the flexbag.

`/HEIGHT`
: `name|client` **/HEIGHT** `height`
Return the height of the given client, which may be specified either by name or by the client itself.

`/POSITION`
: `compass_point name|client` **/POSITION** `xy`
Return the (x,y) position of the indicated compass point on the given client. The client may be specified either by name or by the canvas itself.

`/WIDTH`
: `name|client` **/WIDTH** `width`
Return the width of the given client, which may be specified either by name or by the canvas itself.

/XYADD	`x1 y1 x2 y2` **/XYADD** `x1+x2 y1+y2` Return the vector sum of the two points given.
/XYSUB	`x1 y1 x2 y2` **/XYSUB** `x1-x2 y1-y2` Return the vector difference of the two points given.

see also:

ClassBag

Object

Subclass of **null**
Source File: **class.ps**

This class should be subclassed, even though it can be directly instantiated.

The class Object is the ultimate superclass of every class in the tNt toolkit. Much of the class system's functionality is implemented here.

Direct Methods

/class	-- **/class** `class` Sent to an instance, **/class** returns the class object from which this instance was created.
/class?	-- **/class?** `bool` Return **true** if the object of the send is a class; **false** if it is an instance, or any other NeWS dictionary.
/classinit	-- **/classinit** -- This method is called automatically after class definition, but before any instances are created. Override it if you need to perform per-class initialization that cannot be done as the class is being built.
/classdestroy	`class` **/classdestroy** -- Remove a class from the system. This is a utility,

rather than a method. It is not sent to the class concerned, but consumes it as an argument.

`/classname`

-- **/classname** `classname`
Sent to a class, this method returns the name of the class. sent to an instance it returns the name of the instance's class.

`/cleanoutclass`

-- **/cleanoutclass** --
Clean out a class dictionary for reuse. Whenever a class is redefined this method is called automatically. The effect of this is that instances or subclasses of the old class will immediately start using the new class code.

`/defaultclass`

-- **/defaultclass** `class`
Returns a default subclass if one has been defined, **self** otherwise. A default subclass is considered to be a reasonable subclass to instantiate for normal use. Many instrinsic classes have OPEN LOOK equivalents as their default subclass.

`/descendantof?`

`instance|class` **/descendantof?** `bool`
Test to see if the argument is a descendant of the class or instance being sent to.

`/destroy`

-- **/destroy** --
Destroy this object. This method is called automatically when NeWS detects that there are no more hard references to it. Subclassers should override this method to break any circular reference chains that include a soft reference to the object. When **/destroy** returns, all references to the object should have been removed.

`/destroydependent`

-- **/destroydependent** --
This method is sent to dependent objects (child canvases, and subframes are examples of such objects) in the process of destroying an object. Class Object imposes no semantics on what should happen to the dependent object in this case.

/doit

<args> proc /doit <results>
Execute the given procedure from within the context of the object of the **send.** In particular **self** resolves to that instance, and **super** to its superclass.

/installmethod

name proc /installmethod --
Create or overwrite a method with the given name and given procedural body. This method can be sent either to a class or to an instance. In the former case the result of **/installmethod** is indistinguishable from having defined this method is the class originally.

/instanceof?

obj /instanceof? bool
This method is sent to a class, not an instance. It returns **true** if the given object is an instance of the class.

/name

-- /name name
Return the name of an instance or class. If name was not set by **/setname,** then the name of an instance is the name of its class. Sending name to a class returns the classname unless the **/setname** method has been applied to it.

/named?

-- /named? bool
Returns **true** if the object of the send has had it's name set via the **/setname** method. This method can be sent to either an instance or a class.

/new

-- /new instance
Create an initialized instance of this class. This method should only be sent to a class. It calls **/newobject** followed by **/newinit.**

/newdefault

-- /newdefault instance
Create an instance using whatever **/defaultclass** returns.

/newinit

-- /newinit --
Initialize an instance. Override this method to perform your per-instance initialization, and

consume any arguments that your subclass requires be presented to **/new.**

`/newmagic`

`<creation args> dict` **/newmagic** `instance`
Convert an existing magic dictionary object, into an object to which sends can be performed. The converted magic dictionary becomes an instance of the class to which this message is sent.

`/newobject`

`<creation args>` **/newobject** `instance`
Create an instance of a class, but do not initialize it.

`/obsolete`

-- **/obsolete** --
Handle an instance becoming 'obsolete'. This method will be sent when an instance loses its last hard references. By default, **/obsolete** calls **/destroy** to clean up the remaining soft references.

`/?promote`

`name object` **/?promote** --
Promote object to be an instance variable associated with name only if the argument value is different from an existing instance variable value.

`/promote`

`name object` **/promote** --
Promote object to be an instance variable whose name is the first argument.

`/promoted?`

`name` **/promoted?** `bool`
Return **true** if name is an instance variable, **false** otherwise.

`/setname`

`name` **/setname** --
Set the name of a class or instance.

`/subclasses`

-- **/subclasses** `[class ...]`
Return an array of class objects that are the currently defined immediate subclasses of the class to which this message is sent.

`/superclasses`

-- **/superclasses** `array`
Return an array of class objects which define the

	superclass chain for this class. The array is returned in order, from most distant to most immediate superclass.
`/understands?`	**name /understands?** `bool` Return **true** if name is defined in the context of the instance or class to which this mesasage is sent.
`/unpromote`	**name /unpromote --** Remove a variable from an instance. No error will occur if no such variable existed.

Subclass Methods

`/HandleObsoleteClass`	**-- /HandleObsoleteClass --** Called by **/ObsoleteEventHandler** when a class goes obsolete. By default this calls **/destroyclass**
`/HandleObsoleteInstance`	**-- /HandleObsoleteInstance --** Called by **/ObsoleteEventHandler** when an instance goes obsolete. By default this calls **/obsolete.**
`/HandleObsoleteOther`	**-- /HandleObsoleteOther --** Called by **/ObsoleteEventHandler** when an object that is not a a class or an instance becomes obsolete.
`/ObsoleteEventHandler`	**event /ObsoleteEventHandler --** A callback method that is invoked by a global obsolete event manager when a class object becomes obsolete. Uses the **/Action** value in an event to call one of **/HandleObsoleteClass, /HandleObsoleteInstance** or **/HandleObsoleteOther.**

Class Variables

/ClassName	Class variable used to hold the name of the class.
/DefaultClass	Variable referencing the default subclass of a class. This variable is **self** by default. Referenced by **/defaultclass**.
/ObsoleteEventMgr	A shared event manager whose job it is to call **/ObsoleteEventHandler** when an obsolescence event is received.

OpenLookAbbrButton

Subclass of **OpenLookButton**
Source file: **OLbutton.ps**

This class can be directly instantiated

This class implements OpenLook Abbreviated buttons: the graphic for the button appears to the right of it, rather than inside it. This class is directly instantiable, but is also mixed into OpenLookAbbrButtonStack.

Direct Methods

/callnotify	– **/callnotify** – Call the button's notify procedure unconditionally, with the button itself on the operand stack.
/cleartarget	`object|null` **/cleartarget** – Selectively clear the target used by the **/sendtarget** method. This method is called automatically when the button is destroyed. See *ClassTarget* for more about targets and their uses.
/demo	– **/demo** `instance` Create a sample abbreviated button and leave it on the stack.
/destroy	– **/destroy** – Turn off tracking if it is on, and destroy the button. If the menu was passed in as an array and

constructed on-the-fly by the button then it too will be destroyed. Otherwise it is up to the caller who passed the menu in to ensure its destruction.

`/disable`

– **/disable** –
Stop responding to user actions; set the button's state to **false**, paint the button accordingly, and remove the tracking process when current requests are serviced.

`/enable`

– **/enable** –
Set the button's state to enabled, and display it accordingly with **/PaintEnabledState**.

`/enabled?`

– **/enabled?** `bool`
Return **true** if the button is currently enabled, **false** if it is disabled. When disabled, the button does not respond to user input.

`/graphic`

– **/graphic** `obj`
Return the graphic object that represents the button on screen.

`/location`

– **/location** `x y`
Return the location of the origin of the button relative to the CTM.

`/minsize`

– **/minsize** `width height`
Return the smallest size the button can be and still appear intelligible.

`/move`

`x y` **/move** –
Move the origin of the button to the specified location in the coordinates of the CTM.

`/new`

`thing|graphic menu|array notify parentcanvas` **/new** **`instance`**
The arguments to /new specify:

- a parent canvas of the button.

- a notification procedure or callback, which is executed whenever the SELECT button pressed and released over the canvas. Remember that tracking for an OPEN LOOK button begins when the mouse cursor enters the canvas and the SELECT button is pressed. Notification occurs when the SELECT button is released over the canvas. In order for the button to correctly function this callback must be null. While you may install you own button callback, the resulting behavior is likely to not be OpenLook compliant. The default callback for abbreviated buttons implements the required OpenLook functionality by executing the default selection from the button menu when the SELECT button is released.
- a menu or menu specification.
  If an array is passed in it is assumed to be a specification for a menu that the button should instantiate automatically. It does this by sending **/newdefault** to ClassMenu: [menu specification] framebuffer **/newdefault** ClassMenu send The result should be a a valid menu instance. Note that this interface does not support all the supported interfaces to **/new** in ClassMenu. The menu passed into **/new** (as an instance or a specification) is displayed whenever the MENU button is pressed over the button. If the supplied callback was null the default callback of the menu will be executed when the SELECT button is released over the button.

- a *graphic* or *thing* to be made into a graphic.
  If the object supplied is a thing or a non-terminal graphic, **/CreateGraphic** is called to make an instance of OPEN LOOKButtonGraphic from the argument.

/notifyproc

`-` **/notifyproc** `proc`
Return the button's current notify procedure.

/preferredsize

`-` **/preferredsize** `width height`
Return the preferred or "ideal" size for this button, which by default is the minimum size (**/minsize**).

/sendtarget

`args /method` **/sendtarget** `results`
This method can be invoked within the notifyproc in order to send a message to some other instance. Many clients will not need to use this indirection. See ClassTarget for more details.

/setarrow

`/Left|/Right|/Up|/Down` **/setarrow** `-`
Abbreviated buttons display an arrow that indicates in which direction the menu will be displayed. This method will set the arrow direction to one of four values and ensure that the menu is appropriately positioned when it is displayed.

/setgraphic

`thing|graphic` **/setgraphic** `-`

/setgraphic

`thing|graphic` **/setgraphic** `-`
Convert the argument to a graphic, if necessary, and store it as the button's graphic. Then, invalidate the button to require layout before repainting.

/setnotifyproc

`proc` **/setnotifyproc** `-`
Set the notification procedure for the button.

/settarget

`object` **/settarget** `-`
Set the target used by the **/sendtarget** method. Note that the button itself does not typically

have a callback of its own. Instead, it usually relies upon the button to provide all its callbacks. Thus setting the target of the menu will also set the target of its menu. See ClassTarget for more about targets and their uses.

`/size` — `- /size w h`
Return the width and height of the button relative to the CTM.

`/target` — `- /target object`
Returns the current target used by the **/sendtarget** method. See *ClassTarget* for more about targets and their uses.

see also:

OpenLookButton

OpenLookAbbrButtonStack

Subclass of **OpenLookAbbrButton,OpenLookButtonStack**
Source file: **OLbutton.ps**

This class can be directly instantiated

This class implements OPEN LOOK abbreviated button menus (previously called button stacks).

see also:

OpenLookAbbrButton, OpenLookButtonStack

OpenLookBaseFrame

Subclass of **OpenLookFrame,ClassBaseFrame**
Source file: **OLframe.ps**

This class can be directly instantiated

Most applications will use an instance of this class as their main window on the screen. The methods documented below are those of interest to the user rather than the expert subclasser.

An OPEN LOOK Base Frame is characterized by the following attributes:

- a header area on the top of the frame containing a header label and a menu button,
- an icon subframe, named **/Icon**, into which the frame closes
- a menu shared with the icon subframe
- an footer area at the bottom of the frame, containing messages on the left and/or right sices of the footer
- resize corners that permit the user to initiate resizing the frame by pressing the SELECT button on the resize corners
- a single client canvas in the middle of the frame.

Direct Methods

`/activate`

-- **/activate** --
Activate event management for the frame and any subframes.

`/active?`

-- **/active?** `bool`
Return **true** if event management has been activated on the frame (and subframes).

`/addsubframe`

`name frame` **/addsubframe** --
Add *frame* as a subframe by *name* and activate event management for it.

`/busy?`

-- **/busy?** `bool`
Return **true** if the frame is busy, which means the frame is temporarily not interested in input because it is processing a previous request. The busy state is indicated on screen with the **/busy** cursor, by default the hourglass.

`/callnotify`

-- **/callnotify** --
Call the frame's notification procedure, which

was stored by **/setnotifyproc**. Commonly, the reason for notification is given to the frame using **/setnotifyreason** before calling the notification procedure.

/changed?

-- **/changed?** `bool`
Return the frame's **/Changed?** variable. This can be used by subclassers to indicate some change of state of interest to them.

/client

-- **/client** `client|null`
Return the frame's client canvas (/Client), or **null** if there is none.

/closesubframes

-- **/closesubframes** --
Close all open subframes and store their names internally; **/opensubframes** can later be used to open them.

/deactivate

-- **/deactivate** --
Turn off event management for the frame and subframes.

/demo

-- **/demo** `frame`
This is a demonstration method that will put on screen a simple OPEN LOOK Base Frame.

/destroy

-- **/destroy** --
Destroy the frame and its subframes , deselecting them as necessary.

/destroyfromuser

-- **/destroyfromuser** --
Destroy the frame, allowing the application to intervene if necessary (for example, to ask the user to save files). Override this method to provide special processing when destroying a frame.

/fitclient

`w h` **/fitclient** `w' h'`
Validate the frame if necessary, and then return the size the frame should be for its client to fit into it.

/flashframe

-- **/flashframe** --
Briefly repaint the frame with the focus indication reversed; then repaint the frame as it was. This creates a flashing effect on screen. This method is used, for example, in the menu for OPEN LOOK popup frames to identify the owner of the popup frame.

/flipiconic

-- **/flipiconic** --
Toggle the frame's iconic state. If it is open, close it into an icon; if it is an icon, open it.

/flipselected

sourceframe **/flipselected** --
If the *sourceframe* is currently selected, deselect it and make this frame selected instead.

/flipzoom

-- **/flipzoom** --
Toggle the zoomed state of the frame; if it is enlarged (zoomed) make it normal size and vice versa.

/focus?

-- **/focus?** bool
Return **true** if the frame has the input focus.

/footer

-- **/footer** graphic|thing|null
graphic|thing|null
Return the left and right footers of the frame. The argument returned for each footer can be either a graphic, a **thing** from which a graphic is made, or **null**.

/frameattribute

name **/frameattribute** bool
Return the value of one of the frame attributes. See /new for a list of frame attributes.

/freeze

bool **/freeze** --
Freeze the frame when the argument is **true**, unfreeze when the argument is **false**. This makes the frame insensitive to events except damage and loss of focus or selection (**/Damaged, /LoseFocus, /LoseSelection**).

/freezeall

bool **/freezeall** --
This method freezes (**true**) or unfreezes (false)

all frames in the frame hierarchy of the recipient. If the recipient is a root frame, it and all subframes freeze or unfreeze, as dictated by the argument. If the recipient is a subframe, however, its superframe(s) and their other descendant frames will freeze/unfreeze as well as the recipient and its subframes.

`/gravity`

-- **/gravity** `/UpperLeft|/UpperRight|/LowerLeft|/LowerRight`
Return the gravity variable for the frame. This is the side of the screen where the frame will tend to be placed when it is opened.

`/label`

-- **/label** `graphic|thing|null`
Return the label of the frame, which may be a graphic, a thing from which a graphic is made, or **null**.

`/location`

-- **/location** `x y`
Return the location of the origin of the frame in coordinates of the CTM.

`/map`

-- **/map** --
Map the frame and move it to the top.

`/mapped?`

-- **/mapped?** `bool`
Return **true** if the frame is currently mapped.

`/menu`

-- **/menu** `menu|null`
Return the menu object associated with this frame as set by **/setmenu** and stored in the **/CanvasMenu** variable The menu is displayed whenever the MENU button is pressed with the pointer over the frame. Return **null** if there is no menu set.

`/minsize`

-- **/minsize** `minwidth minheight`
Return the minimum size for this frame based on the minumum size its main client (**/Client**) requires.

When subclassing, override this method if the calculations do not require the current canvas to be the frame. (If the calculations do require the frame as the current canvas, override /MinSize instead.)

/move

x y /move --
Move the origin of the frame to the specified location in the coordinates of the CTM.

/new

clientcanvas|null AVList parentcanvas /new instance
clientclass AVList parentcanvas /new instance
[args clientclass] AVList parentcanvas /new instance
This method creates a new frame, calling **/newinit** to perform initialization once the instance is created. The arguments to **/new** are:

- the parent canvas of the frame being created
- an attribute-value list (*AVList*), which is an array or dictionary of the attributes of the frame and their initial values. By default, an OPEN LOOK base frame has the **/Close**, **/Footer**, **/Label**, and **/Reshape** attributes set to **true** and **/Pin** set to **false**. To create an instance with different attribute values, the AV-list would be of the form **[attribute bool]**, for example **[/Reshape false]**. The contents of the AV-list may be **null**, in which case, specify **[]**. A dictionary can also be used to specify the AV list, with the attribute names as keys.

  dictbegin /Footer false def /Reshape false def dictend

- the client,
  either the client canvas itself, or the class from which to create an instance to be the client, or **null**. Specify the client as an argument if it has already been instantiated when the frame is created. Specify **null** if the client is to be added at a later time. Specify a class as the client argument to have this method create an instance of that class as the client. If no additional processing is done on the client between its creation and its being added to the frame, it is more efficient to specify a class than an instance. Finally, if additional arguments are required to create an instance, group them into an array, following the *clientclass*.

`/newinit`

`--` **/newinit** `--`
Process the arguments to /new, and create an icon subframe.

`/notifyproc`

`--` **/notifyproc** `proc`
Return the frame's notification procedure.

`/notifyreason`

`--` **/notifyreason** `keyword`
Return the reason the frame's notification procedure was called. This information was saved by sending **/setnotifyreason** to the frame before sending

`/notifyselected`

`obj proc` **/notifyselected** `--`
This method is sent to a class to have *proc* sent to all selected instances of that class.

`/open`

`bool` **/open** `--`
Open or close the frame and its subframes depending on the sense of the argument: open if **true**, close if **false**.

`/opened?`

`--` **/opened?** `bool`
Return **true** if this frame is open.

`/opensubframes`

`--` **/opensubframes** `--`
Open the subframes that had previously been closed and stored using **/closesubframes.**

`/owner`

`--` **/owner** `graphic|thing|null`
Returns the portion of the frame's label that indicates the frame's owner (superframe).

`/pin`

`--` **/pin** `--`
Pin the frame, if it is pinnable. This method invokes the pin notification procedure, **/PinNotify.**

`/pinned?`

`--` **/pinned?** `bool`
Returns **true** if the frame is currently pinned.

`/place`

`--` **/place** `--`
This method computes a default location for the frame and places it there. If the frame has already been sized, it keeps that size. Otherwise, the frame assumes its preferred size.

`/popuphelp`

`object` **/popuphelp** `--`
Create and pop up a help frame and display help about the specified object. If sent to a subframe, this message is passed on to the frame's superframe. The help frame is created as a subframe with the name /Help. If a help frame has already been created, it is reused.

`/preferredsize`

/preferredsize `preferredwidth preferredheight`
Return the frame's preferred size, which is based on the preferred size of its main client, **/Client**.

`/removesubframe`

`name` **/removesubframe** `oldsubframe true`
`name` **/removesubframe** `false`
Remove the *name* subframe, if it exists, returning the subframe and **true**. If the subframe does not exist, return **false**. Note that the frame is not deactivated by this method.

`/reshape`

`x y w h` **/reshape** `--`
Reshape the frame to fit the given coordinates

and invalidate the frame. This forces the frame and its clients to do layout before repainting.

`/reshapefromuser`

-- **/reshapefromuser** --
This method is used by an application to initiate user interaction to reshape the frame. First, the cursor will change, by default to a crosshair. Next, the NeWS process will block until the user drags the mouse to sweep out a bounding box. Then, the frame will be resized to fit the bounding box.

`/rootframe`

-- **/rootframe** `frame`
Returns the root frame of this frame. A root frame is a frame that has no superframe.

`/selected?`

-- **/selected?** `bool`
Returns **true** if this frame is the selection target.

`/selectedframes`

-- **/selectedframes** `[selectedframes]`
This method is sent to a subclass of ClassFrame and returns an array of its instances that are selected.

`/sendrootframe`

`<methodargs> /method` **/sendrootframe** --
Send *method* together with the supplied arguments to the frame's root frame.

`/sendselected`

`/method|array` **/sendselected** --
This method is sent to a subclass of ClassFrame to send the arguments to each selected instance of that class. The arguments may be a method name or an array of the form **[arg arg ... /method]**. The array may be executable or not.

`/sendsubframe`

`args... /method subframe` **/sendsubframe** --
Send *method* and *args* to *subframe*. If the subframe does not exist, an error occurs.

`/sendsuperframe`

`methodargs /method` **/sendsuperframe** --
Send */method* and *methodargs* to the frame's super frame.

/setbusy

`bool` **/setbusy** --
Mark the frame as busy (**true**) or not busy (**false**).

/setchanged

`bool` **/setchanged** --
Mark the frame as "changed". This facility is provided for subclassers or applictions that need to maintain additional state for their frames.

/setclient

`newclient` **/setclient** `oldclient|null`
Set the frame's client to the specified canvas. Return the previous client, or null if there was none. or **null** if there was none.

/setfocus

`bool` **/setfocus** --
Give the frame the input focus.

/setfooter

`Lgraphic|Lthing|null`
`Rgraphic|Rthing|null` **/setfooter** --
Set the objects that define the right and left footer elements. Each element can be **null**, a graphic, or a **thing** from which a graphic is made. Null does not change the existing element, if there is one. To remove an element, set it to **nullstring**.

/setframeattribute

`name bool` **/setframeattribute** --
`array` **/setframeattribute** --
`dict` **/setframeattribute** --
This method is used to dynamically change frame attributes, possibly resulting in the frame requiring validation afterwards. The arguments are of the form *attribute bool*. If there are several argument pairs, they may be enclosed in an array:

[*att1 bool att2 bool2*]

Alternatively, the argument-value pairs may be enclosed in a dictionary:

dictbegin *att1 bool1* def *att2 bool2* def dictend

/setgravity

`/UpperLeft|/UpperRight|/LowerLeft|/LowerRight`
`/setgravity --`
Set the frame's gravity, which is the side of the

screen the frame will tend toward when first opened.

`/seticon`

`{paint}|[array]|canvas|(string)|/name` **/seticon** --

Create the frame's icon as specified:

{paint}	paint procedure of the icon canvas
[args..class]	an instance is created and used for the icon
canvas	use this canvas as the icon image
(string)	use the string as the icon image
/name	get character from iconfont

`/seticongravity`

`/Left|/Right|/Top|/Bottom` **/seticongravity** --

Set the gravity variable for the frame's icon.

`/seticonlabel`

`(string)` **/seticonlabel** --
Set the label for the frame's icon, removing the old label if there was one. Use **nullgraphic** for no label.

`/setlabel`

`graphic|thing|null` **/setlabel** --
Store a new value for the frame's label: a graphic, a **thing** from which a graphic is made, or **null**.

`/setnotifyproc`

`proc` **/setnotifyproc** --
Set the notification procedure for the frame. This procedure is called with the frame itself on the operand stack whenever a user action changes the state of the frame.

`/setnotifyreason`

`keyword` **/setnotifyreason** --
This method is used to supply a reason for calling the notification procedure, just before calling it.

`/setowner`

`graphic|thing|null` **/setowner** --
Set the portion of the frame's label that indicates the frame's owner (superframe).

`/setproperties`

-- **/setproperties** --
If a properties subframe already exists for this

frame, open it. Otherwise, create one and open it. The properties frame is shared.

`/size`

-- **/size** `w h`
Return the current width and height of the frame.

`/subframe`

name **/subframe** `frame true`
name **/subframe** `false`
Return the named subframe and **true** if the subframe exists, otherwise return **false**. This can be used to retrieve the icon (**/Icon**), for example.

`/subframe?`

name **/subframe?** `bool`
Returns **true** if subframe *name* exists.

`/subframes`

-- **/subframes** `dict`
Return a dictionary whose keys are the names of subframes of this frame, and whose values are the subframe instances.

`/superframe`

-- **/superframe** `frame`
Return this frame's superframe or this frame itself if it has no superframe.

`/toback`

-- **/toback** --
Move the frame and its subframes to the bottom (back) on the screen.

`/tofront`

-- **/tofront** --
Move the frame and its subframes to the top (front) on the screen.

`/unfitclient`

`w h` **/unfitclient** `w' h'`
Return the size the frame's client should be to fit into the frame. This is the opposite of /fitclient, which says how big the frame should be to hold the client. Note that, if necessary, the frame is validated first.

`/unmap`

-- **/unmap** --
Unmap the frame and give up being selected.

`/unpin`

-- **/unpin** --
If the frame is pinned, unpin it.

/zoom

`bool` **/zoom** --
Expand the frame to full size if the argument is **true**, or revert to normal size if **false**.

/zoomed?

-- **/zoomed?** `bool`
Return **true** if the frame is currently full size.

see also:

OpenLookFrame, ClassBaseFrame

OpenLookButton

Subclass of **ClassButton**
Source file: **OLbutton.ps**

This class can be directly instantiated.

This class implements OpenLook buttons. It is also the main superclass for button stacks and abbreviated buttons.

Direct Methods

/callnotify

- **/callnotify** -
Call the button's notify procedure unconditionally, with the button itself on the operand stack.

/cleartarget

`object|null` **/cleartarget** -
Selectively clear the target. If null is given, the target is cleared. If *object* is given, the target is cleared only if the *object* and the traget are the same. This ensures that the target cannot be incorrectly cleared.

Note that the **/destroy** method of any object pointed to by a target should clear the target as part of destroying the object.

/default?

- **/default?** `bool`
Return **true** if this button has been designated the default (for a menu, for example)

`/demo`

– /demo `instance`
Show a sample OPEN LOOK button, whose call-back writes to the console.

`/destroy`

– /destroy –
Turn off tracking if it is on, and destroy the button.

`/disable`

– /disable –
Stop responding to user actions; set the button's state to **false**, paint the button accordingly, and remove the tracking process when current requests are serviced.

`/enable`

– /enable –
Set the button's state to enabled, and display it accordingly with **/PaintEnabledState**.

`/enabled?`

– /enabled? `bool`
Return **true** if the button is currently enabled, **false** if it is disabled. When disabled, the button does not respond to user input.

`/graphic`

– /graphic `obj`
Return the graphic object that represents the button on screen.

`/location`

– /location `x y`
Return the location of the origin of the button relative to the CTM.

`/minsize`

– /minsize `w h`
Return the smallest size the button can be and still appear intelligible.

`/move`

`x y` **/move –**
Move the origin of the button to the specified location in the coordinates of the CTM.

`/new`

`thing|graphic notifyproc`
`parentcanvas` **/new** `instance`
The arguments to **/new** specify:

- a parent canvas of the button.
- a notification procedure or callback, which is executed whenever the SELECT button on the mouse is released over the button. Remember that tracking for an OPEN LOOK button begins when the mouse cursor enters the button and the SELECT button is pressed. Notification occurs when the SELECT button is released over the button.
- a *graphic* or *thing* to be made into a graphic.
  If the object supplied is a thing or a non-terminal graphic, **/CreateGraphic** is called to make an instance of OpenLookButton-Graphic from the argument.

`/notifyproc`

`–` **/notifyproc** `proc`
Return the button's current notify procedure.

`/preferredsize`

`–` **/preferredsize** `width height`
Return the preferred or "ideal" size for this button, which by default is the minimum size (**/minsize**).

`/sendtarget`

`args /method` **/sendtarget** `results`
Send the method and arguments to the target object. Any button can have a "target" which you can set, get, and send to. For example, if you have a button that needs to send a message to object *foo* when the button's callback is executed:

```
/b (Hello) {/foo-method /sendtarget 3 -1 roll send}
framebuffer /new OpenLookButton send def
...
foo /settarget b send
```

When the button is pressed, *foo-method* will be sent to *foo*. You can change the target at any time if you want to make your button send to different objects at different times.

An error results from sending anything to a null target

`/setdefault`

`bool` **/setdefault** –
Mark this button as a default. Currently this only affects the graphical look of the button by drawing an extra "default ring" drawn in it.

`/setgraphic`

`thing|graphic` **/setgraphic** –
Convert the argument to a graphic, if necessary, and store it as the button's graphic. Then, invalidate the button to require layout before repainting.

`/setnotifyproc`

`proc` **/setnotifyproc** –
Set the button's notify procedure, overwriting the previous one.

`/settarget`

`object` **/settarget** –
Set the target object, overwriting the previous target, if there was one.

`/size`

– **/size** `w h`
Return the width and height of the button in the coordinates of the CTM.

`/target`

– **/target** `object`
Return the target of this button. If left unset, this value defaults to the button itself. See ClassTarget for a more complete description of targets and their usage in controls and menus.

Class Variables

`/Default`

An OPEN LOOK button has an additional binary state named **/Default**, which is not connected with its enabled/disabled state or its on/off value. It is used, for example, in OPEN LOOK button menus to designate one button as the default selection for the menu. The button designated as the default is painted with an ring inside its outline. The default value of this variable is **false**.

see also:

ClassButton

OpenLookButtonStack

Subclass of **OpenLookButton**
Source file: **OLbutton.ps**

This class can be directly instantiated.

This class implements OpenLook Button Stacks (now called Button Menus in OPEN LOOK), as well as being the primary superclass for OpenLook Abbreviated Button Stacks.

Direct Methods

`/demo`

– **/demo** `instance`
Create a demonstration button stack on the framebuffer. This method will only be available if **/IncludeDemos?** was **true** at the time this class was read in.

`/destroy`

– **/destroy** –
Turn off tracking if it is on, and destroy the button. If the menu was passed in as an array and constructed on-the-fly by the button then it too will be destroyed. Otherwise it is up to the caller who passed the menu in to ensure its destruction.

`/disable`

– **/disable** –
Stop responding to user actions; set the button's state to **false**, paint the button accordingly, and remove the tracking process when current requests are serviced.

`/enable`

– **/enable** –
Set the button's state to enabled, and display it accordingly with **/PaintEnabledState.**

`/enabled?`

– **/enabled?** `bool`
Return **true** if the button is currently enabled,

false if it is disabled. When disabled, the button does not respond to user input. When enabled it will respond to both the SELECT and MENU mouse buttons.

`/graphic`

– **/graphic** `obj`
Return the graphic object that represents the button on screen.

`/location`

– **/location** **x y**
Return the location of the origin of the button relative to the CTM.

`/menu`

– **/menu** **menu**|`null`
Return the menu object associated with this button as set by **/new** or **/setmenu**. The menu is displayed whenever the MENU button is pressed with the pointer over the canvas. The menu default is executed if the button callback is null and SELECT button is pressed and released over the canvas. Return **null** if there is no menu set.

`/menubelow`

`bool` **/menubelow** –
This method is used to determine on which side of the button the menu will pop-up on. By default the menu is displayed below the button stack. passing **false** to this method would cause the menu to pop-up to the right of the button. The functionality to position the menu to the right of the button has not yet been implemented so only **true** should be passed to this method for now.

`/menubelow?`

– **/menubelow?** `bool`
Returns **true** of the menu for this button will be displayed below the button. Othertwise the menu will come up to the right of the button.

`/move`

`x y` **/move** –
Move the origin of the button to the specified location in the coordinates of the CTM.

/new

```
thing menu|array notifyproc
parentcanvas /new  instance
```

The arguments to **/new** specify:

- a parent canvas of the button.
- a notification procedure or callback, which is executed whenever the SELECT button pressed and released over the canvas. Remember that tracking for an OPEN LOOK button begins when the mouse cursor enters the canvas and the SELECT button is pressed. Notification occurs when the SELECT button is released over the canvas. In order for the button to correctly function this callback must be null. While you may install you own button callback, the resulting behavior is likely to not be OPEN LOOK compliant. The default callback for abbreviated button stacks implements the required OpenLook functionality by executing the default selection from the button menu when the SELECT button is released.
- a menu or menu specification.
  If an array is passed in it is assumed to be a specification for a menu that the button should instantiate automatically. It does this by sending **/newdefault** to ClassMenu: [menu specification] framebuffer **/newdefault** ClassMenu send The result should be a a valid menu instance. Note that this interface does not support all the supported interfaces to **/new** in ClassMenu. The menu passed into **/new** (as an instance or a specification) is displayed whenever the MENU button is pressed over the button. If the supplied callback was null the default callback of

the menu will be executed when the SELECT button is released over the button.

- a *graphic* or *thing* to be made into a graphic.
  If the object supplied is a thing or a nonterminal graphic, **/CreateGraphic** is called to make an instance of OPEN LOOK ButtonGraphic from the argument.

`/newinit`

`thing menu|array notify` **/newinit** –
This method is called by **/new** and is responsible for consuming the callback, the menu speicification, and the thing or graphic. This is primarily a method used by subclassers; users should not call this method directly.

`/preferredsize`

– **/preferredsize** `width height`
Return the preferred or "ideal" size for this button, which by default is the minimum size (**/minsize**).

`/setgraphic`

`thing|graphic` **/setgraphic** –
Convert the argument to a graphic, if necessary, and store it as the button's graphic. Then, invalidate the button to require layout before repainting.

`/setmenu`

`menu|array` **/setmenu** –
Install or remove a popup menu for this canvas. The menu is activated by the user pressing the MENU button when the pointer is over the button. The menu defauilt is activated by by the user pressing and releasing the SELECT button when the pointer is over the button. If an array is passed in it is assumed to be a specification for a menu that the button should instantiate automatically. It does this by sending **/newdefault** to ClassMenu:

[menu specification] framebuffer /newdefault ClassMenu send

The result should be a a valid menu instance. Note that this interface does not support all the supported interfaces to **/new** in ClassMenu.

`/size`

– **/size** w h

Return the width and height of the button relative to the CTM.

Subclass Methods

`/DisplayDefault`

– **/DisplayDefault** –

This method displays the default menu selection in place of the button graphic. Typically this method is called automatically when the user presses the SELECT mouse button over an active, enabled button stack.

`/UnDisplayDefault`

– **/UnDisplayDefault** –

This method displays the button graphic in place of the default menu selection .

see also:

OpenLookButton

OpenLookCheckBox

Subclass of **OpenLookXSettingControl**
Source file: **OLxctrl.ps**

This class can be directly instantiated.

This class implements OPEN LOOK check boxes, a nonexclusive setting used in lists of yes-no choices. The implementation of this class is subject to change in the future.

see also:

OpenLookXSettingControl

OpenLookChoggle

Subclass of **OpenLookXSetting**
Source file: **OLxset.ps**

This class can be directly instantiated.

This classs implements OpenLook "choggles", which is a set of items, of which zero or one is selected at any time. It can be thought of as an exclusive setting that can be turned off altogether.

see also:

OpenLookXSetting

OpenLookCommandFrame

Subclass of **OpenLookFrame,ClassCommandFrame**
Source file: **OLframe.ps**

This class can be directly instantiated.

This class is OpenLookBaseFrame with the following differences; command frames:

- have a pin
- do not have a footer
- cannot be closed into an icon (have no "iconic" state)
- cannot be quit from or closed into an icon (pressing SELECT over the close box or selecting "Dismiss" from their menu simply unmaps them from the screen.)

see also:

OpenLookBaseFrame

OpenLookFrame

Subclass of **ClassFrame**
Source file: **OLframe.ps**

This class should be subclassed rather than directly instantiated.

This class is used to "mix-in" OpenLook features into frame subclasses. For example, it is mixed in with ClassBaseFrame to produce OpenLookBaseFrame. Likewise, it is mixed in to produce the other OPEN LOOK frame types: command, icon, help, notice, and property.

OpenLookHelpFrame

Subclass of **OpenLookFrame,ClassHelpFrame**
Source file: **OLframe.ps**

This class can be directly instantiated.

This class is OpenLookBaseFrame with the following differences; Help frames:

- have a pin
- do not have a footer
- do not have a resize tab.
- cannot be quit or closed into an icon (pressing SELECT over the close box or selecting 'Dismiss" from their menu simply unmaps them from the screen).

see also:

OpenLookBaseFrame

OpenLookHorizontalScrollbar

Subclass of **ClassScrollbar,ClassBag**
Source file: **OLsbar.ps**

This class can be directly instantiated

The only differences between this class and OpenLookVerticalScrollbar are:

1. the vertical scrollbar has an origin in the upper left corner, whereas OpenLookHorizontalScrollbar uses the standard lower left origin.
2. The menu associated with the vertical scrollbar has "Here to top", whereas OpenLookHorizontalScrollbar has "Here to left".

see also:

OpenLookVerticalScrollbar, ClassScrollbar, ClassBag

OpenLookHorizontalSlider

Subclass of **ClassDialControl**
Source file: **OLslider.ps**

This class can be directly instantiated

An OpenLookHorizontalSlider is an analog control. Users can pick up and drag the drag box, or click to the left or right of it. This class has the same interface as OpenLookVerticalSlider.

Direct Methods

`/destroy`

-- **/destroy** --
Destroys the slider drag box and then destroys itself.

`/minsize`

-- **/minsize** `minwidth minheight`
Returns the minimum width and height of the slider. This leaves enough room for the drag box only.

`/motion`

-- **/motion** `motion`
Returns the current motion, as set by **/SetMotion.** The motion for sliders is either **/Absolute,** when the drag box is dragged, or **/Line,** when the mouse is clicked outside the drag box.

`/new`

`notifyproc parentcanvas` **/new** `instance`
Creates and returns an instance of a slider. The

parentcanvas is consumed by **/newobject.** **/newinit** consumes the notifyproc.

`/newinit`

`notifyproc` **/newinit** --
Initializes the slider. This does the normal control initialization and then creates the slider drag box.

`/setdelta`

`name value` **/setdelta** --
This sets the specified delta to the value. The only delta that is used by sliders is **/Line.**

`/setnormalization`

`number` **/setnormalization** --
The normalization defines the granularity of the values for the slider.

`/setnotifyproc`

`proc` **/setnotifyproc** --
Takes the argument proc and makes it the callback for the slider. The callback will be called by **/checknotify** with the slider itself on the stack.

`/setposition`

`event` **/setposition** --
Expects an X/Y pair or an event (which it then turns into an X/Y pair) and sets the value of the slider according to that specified position.

`/setrange`

`min max` **/setrange** --
Sets the range displayed by the slider.

Subclass Methods

`/ClientDown`

`event` **/ClientDown** --
This method is called when the user mouses down over the slider. This sets the slider motion according to where the mouse hit occurred. If the mouse hit occurs inside the slider drag box, the motion is **/Absolute,** otherwise it is **/Line.**

`/ClientDrag`

`event` **/ClientDrag** --
This drags the drag box if the motion is **/Absolute,** that is, if the last **/ClientDown** was inside

the drag box. This calls **/checknotify.** If you don't want notification on every drag event, override **/CallNotify?** to return **false** when the current motion is **/Absolute.**

`/ClientUp`

`event` **/ClientUp** --
This method is called when the user lets go of the mouse button that caused **/ClientDown** to be called. If the motion was **/Line** then the drag box is moved now. If the motion was **/Absolute** the drag box is already positioned. **/checknotify** is called in either case.

`/PaintCable`

-- **/PaintCable** --
Paints the slider cable.

`/PaintCanvas`

-- **/PaintCanvas** --
Paints the slider with the current value.

`/SetMotion`

`event` **/SetMotion** --
Sets the motion of the slider to be **/Absolute** when the drag box is dragged, and to **/Line** when the mouse is clicked outside the drag box (but otherwise inside the slider).

see also:
ClassDialControl

OpenLookIconFrame

Subclass of **OpenLookFrame,ClassIconFrame**
Source file: **OLframe.ps**

This class can be directly instantiated.

An OpenLookIconFrame is automatically created to accompany every Open-LookBaseFrame; it is a full blown frame that by default has no resize corners, footer, or pin.

see also:
OpenLookBaseFrame

OpenLookMenu

Subclass of **ClassMenu**
Source file: **OLmenu.ps**

This class can be directly instantiated

This class implements pop-up and stay-up pinnable OpenLook menus. It is meant to be directly instantiated. Also note that ClassCanvas provides code to watch for the menu button, and show any menu registered via /setmenu.

Direct Methods

/change	`location thing|graphic genproc|sublist|null proc|null` **/change** – Replace the menu item at the specified location. The arguments other than location are the same as for /new. This method invalidates the menu, so it must be repainted after the change.
/cleartarget	`object|null` **/cleartarget** – Clear the menu target. If an object is supplied as a parameter, the target will be cleared only if the current target matches the parameter. See ClassTarget for more about targets and their uses.
/default	– **/default** `index` Return the index of the default menu item.
/delete	`location` **/delete** – Delete the menu item at the specified location. This invalidates the menu, so it should be repainted after the change.

`/demo`

– /demo canvas
Invoke a demonstration menu. The purpose of this menu is to show how to use some commonly used methods of Open Look menus.

`/destroy`

– /destroy –
Destroy the menu and its pinned copy if present.

`/disableitem`

index /disableitem –
Disables a menu item rendering it unselectable. The item's color is changed to DisabledColor to indicate its disabled state.

`/doaction`

– /doaction –
Execute the action associated with the currently selected menu item, i.e. the item whose index matches the menu's current value. If the item has a notify proc, it is called. If the item has a submenu, the notify proc of the submenu's default item is called.

`/dodefault`

– /dodefault –
Execute the action associated with the default menu item. The menu's value is set to its default, and /doaction is called. This method does nothing if the menu has no default or if the default item is disabled.

`/enableitem`

index /enableitem –
Enable the menu item at the specified location.

`/graphic`

location /graphic graphic
Return the graphic associated with the menu item at the specified location.

`/insert`

location graphic procedure|menu|null proc /insert –
Insert a new menu item at the specified location. Except for location, the arguments are the same as for /new. This method invalidates the menu, so it must be repainted after the change.

/itemcount	- **/itemcount** `integer` Return the number of items in the menu.
/itemenabled?	`location` **/itemenabled?** `bool` Return true if the item at the specified location is enabled.
/label	- **/label** `graphic|null` Return the graphic associated with the menu's label.
/layout	- **/layout** `width height` Arrange the menu label, pin and items according to the layout style specified by /setlayoutstyle. Returns the size of the menu.
/layoutstyle	- **/layoutstyle** `RowMajor? Rows|null Columns|null` Return the layout style for the menu. See class RowColumnLayout.
/maxlocation	- **/maxlocation** `location` The largest location of the menu. Menu locations are specified as small integers ranging from 0...n-1, where n is the total number of control items in the menu.

/new	`[thing|graphic genproc|sublist|null proc|null ...] parent` **/new** `inst`
	`thing|graphic ...] genproc|null proc|null parent` **/new** `inst`
	Create a menu. See the *OpenLook Menu* section for a description of the parameters.

/newinit	`[thing|graphic genproc|sublist|null proc|null ...]` **/newinit** - `[thing|graphic ...] genproc|null proc|null` **/newinit** -
/nonxvalue	- **/nonxvalue** `[nonxvalues...]|null` Return an array of nonexclusive menu items currently selected. Returns an empty array if no nonexclusive items are selected. Returns null if the menu contains no nonexclusive items.

/notifyproc

`location` **/notifyproc** `procedure|null`
Return the notify proc associated with a particular menu item.

/owner

`-` **/owner** `string|null`
Return the owner string or graphic associated with the menu.

/pin

`-` **/pin** `-`
Pin a pinnable menu.

/pinnable?

`-` **/pinnable?** `bool`
Return **true** if the menu is pinnable.

/pinned?

`-` **/pinned?** `bool`
Return **true**true if the menu is pinned.

/popdown

`-` **/popdown** `-`
Remove the menu and any submenus from the screen.

/popup

`x y` **/popup** `-`
Pop up the menu at the specified location, relative to the CTM. The menu is activated.

/searchgraphic

`graphic` **/searchgraphic** `index true`
`graphic` **/searchgraphic** `false`
`graphic startloc` **/searchgraphic** `index true`
`graphic startloc` **/searchgraphic** `false`
Search for the menu item having the given graphic. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/searchthing

`thing` **/searchthing** `index true`
`thing` **/searchthing** `false`
`thing startloc` **/searchthing** `index true`
`thing startloc` **/searchthing** `false`
Search for the menu item having the given thing. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/sendtarget

`args /method` **/sendtarget** `results`
This method can be invoked within the notifyproc in order to send a message to some other instance. See ClassTarget for more details.

/setdefault

`index` **/setdefault** –
Set the default menu item.

/setgraphic

`location thing|graphic` **/setgraphic** –
Set or change the graphic at the specified menu item location.

/setlabel

`null|graphic|thing` **/setlabel** –
Set or change the label at the top of the menu.

/setlayoutstyle

`RowMajor? Rows|null Columns|null` **/setlayoutstyle** –
Change the layout style of the menu. See class RowColumnLayout.

/setnotifyproc

`location procedure|null` **/setnotifyproc** –
Change the notification procedure (callback) associated with a menu item. Note that if you specify a single notify proc when creating a menu, the notify proc is duplicated for each menu item. You may therefore change individual menu item callbacks without affecting others. Also, to change the callbacks for all menu items, you must call **/setnotifyproc** for each item in the menu.

/setowner

`string|null|proc` **/setowner** –
Set the owner field of a menu. When the menu is pinned, the owner field is shown to the left of the menu label. This helps users distinguish otherwise identical coexisting pinned menus. The owner is a string that may either be specified explicitly or may be returned by a piece of executable code (the proc).

/setpinnable

`bool` **/setpinnable** –
Set the menu to be pinnable. By default a menu does not have a pin.

/setsublist

`location genproc|sublist|null` **/setsublist** -

Set the submenu associated with a menu item. The submenu may either be a menu instance or may be a procedure that returns a menu instance.

/settarget

`object` **/settarget** -

Set the target used by the /sendtarget method. See ClassTarget for more about targets and their uses.

/setvalue

`index|null` **/setvalue** -

Set the current value of the menu.

/showat

`x y` **/showat** -

`event` **/showat** -

Draw the menu at specified x and y locations and start a tracking manager for the menu. Showat has the same effect as pressing the menu button.

/sublist

`location` **/sublist** `sublist|null`

Get the submenu associated with a menu item. Returns null if there is no submenu for the item.

/sublist?

`location` **/sublist?** `boolean`

Return true if the specified menu item has an associated submenu.

/target

\- **/target** `object`

Return the target object.

/value

\- **/value** `index`

Return the last selected menu item. The value is an integer from 0 to one less than the number of items in the menu. The value will be **null** if the menu is displayed, but no item is selected.

/valuething

\- **/valuething** `thing|null`

Returns the thing associated with the current value.

/xvalue

- **/xvalue** `index|null`
Return the selected exclusive menu item. Returns **null** if the menu contains no exclusive menu items or if none are selected.

OpenLookNonXSetting

Subclass of **OpenLookXSetting**
Source file: **OLxset.ps**

This class can be directly instantiated.

This class implements an Open Look nonexclusive setting group. The group displays multiple setting items on a single canvas. The group acts like a control in that it has a single value, although since more than one setting item may be selected at once, the value is an array of all selected values. Each item has an associated notify procedure that is called when the item is selected or deselected.

Direct Methods

/adjust

`location` **/adjust** -
Toggle the value of the item indicated by *location*. If *location* is greater than the number of items, toggle the last one.

/border

- **/border** `number`
Return the size of the border around the entire group of settings.

/change

`location thing|graphic genproc|sublist|null proc|null` **/change** -
Change the setting item at the specified location. The parameters, other than location, are the same as for **/new**.

/cleartarget

`object|null` **/cleartarget** -
Clear the control's target object. If an object is supplied as a parameter, the target will be

cleared only if the current target matches the parameter. See *ClassTarget* for more about targets and their uses.

`/clientcount`
– **/clientcount** `int`
Return the number of items in the setting.

`/clientlist`
– **/clientlist** `array`
Return an array of dictionaries, one per item in the control group.

`/delete`
`location` **/delete** –
Delete the specified setting item from the group.

`/demo`
– **/demo** `instance`
Create a demonstration nonexclusive setting on the framebuffer.

`/destroy`
– **/destroy** –
Destroy the exclusive setting control.

`/disableitem`
`index` **/disableitem** –
Disable the item at the specified location.

`/doaction`
– **/doaction** –

`/enableitem`
`index` **/enableitem** –
Enable the item at the specified location.

`/gaps`
– **/gaps** `horizontalgap verticalgap`
Return the amount of space between the setting items.

`/graphic`
`location` **/graphic** `graphic`
Return the graphic for the specified setting item.

`/insert`
`location thing|graphic genproc|sublist|null proc|null` **/insert** –
Insert a new setting item at the specified position in the group. The parameters, other than location, are the same as for **/new**.

`/invalidate`
– **/invalidate** –
Invalidate the setting and all of its item graphics.

/itemcount	`-` **/itemcount** `integer` Return the number of items in the setting group.
/itemenabled?	`location` **/itemenabled?** `bool` Return **true** if the specified item is enabled.
/layout	`-` **/layout** `-` Lay out the setting items. This sizes and places the items according to the layout style for the setting.
/layoutstyle	`-` **/layoutstyle** `RowMajor? Rows|null Columns|null` Return the layout style for the menu. See *RowColumnLayout*.
/location	`-` **/location** `x y` Return the location of the setting canvas.
/maxlocation	`-` **/maxlocation** `location` The location of the last item in the control. Item locations range from 0 to **/maxlocation. /maxlocation** is one less than **/itemcount.**
/minsize	`-` **/minsize** `minwidth minheight` Return the minimum size of the setting.
/move	`x y` **/move** `-` Move the setting's canvas to the specified location, relative to the CTM.

/new	`[thing|graphic proc|null ...]`
	`canvas` **/new** `instance`
	Create a nonexclusive setting group. Refer to the introductory section *OPEN LOOK Settings* for the details of the arguments.

/notifyproc	`location` **/notifyproc** `procedure|null` Return the notify proc for the item at the specified location.
/preferredsize	`-` **/preferredsize** `width height` Return the preferred size of the setting group.

/reshape	`x y w h` **/reshape** – Reshape the setting group's canvas, and invalidate the setting group.

/searchgraphic	`graphic` **/searchgraphic** `index true`
	`graphic` **/searchgraphic** `false`
	`graphic startloc` **/searchgraphic** `index true`
	`graphic startloc` **/searchgraphic** `false`
	Search for the setting item having the given graphic. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/searchthing	`thing` **/searchthing** `index true`
	`thing` **/searchthing** `false`
	`thing startloc` **/searchthing** `index true`
	`thing startloc` **/searchthing** `false`
	Search for the setting item having the given thing in its graphic. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/sendtarget	`args /method` **/sendtarget** `results` Send a message to the target object. Typically used in the control's notify proc. See ClassTarget for more details.
/setborder	`number` **/setborder** – Set the size of the border around the entire group of settings.
/setgaps	`horizontalgap|null` `verticalgap|null` **/setgaps** – Change the amount of space between the setting items. A null parameter leaves that value unchanged.
/setgraphic	`location thing|graphic` **/setgraphic** – Change the graphic for the specified setting item.

`/setlayoutstyle`

`RowMajor? Rows|null Columns|null` **/setlayoutstyle** –
Change the layout style of the menu. See *RowColumnLayout* for a more complete description of the arguments.

`/setnotifyproc`

`location procedure|null` **/setnotifyproc** –
Change the notification procedure (callback) associated with a setting item. Note that if you specify a single notify proc when creating the setting group, the notify proc is duplicated for each item. You may therefore change individual item callbacks without affecting others. Also, to change the callbacks for all items, you must call **/setnotifyproc** for each item.

`/settarget`

`object` **/settarget** –
Set the target used by the **/sendtarget** method. See *ClassTarget* for more detail about targets and their uses.

`/setvalue`

`index|[index index ...]` **/setvalue** –
Change the value of the setting. If a single index is supplied, that item is selected and all other items are deselected. If an array is supplied, the items specified by the indices in the array are selected.

`/target`

– **/target** `object`
Return the target object. See ClassTarget.

`/value`

– **/value** `index`
Return an array of indices corresponding to selected items. If no items are selected, an empty array is returned.

`/valuething`

– **/valuething** `[things]`
Return the *thing* of the graphic of the selected item, or null.

see also:

OpenLookXSetting, RowColumnLayout, ClassTarget

OpenLookNoticeFrame

Subclass of **OpenLookFrame,ClassNoticeFrame**
Source file: **OLframe.ps**

This class can be directly instantiated.

This class is OpenLookBaseFrame with the following differences; command frames:

- have a pin
- do not have a footer
- cannot be closed into an icon (have no "iconic" state)
- resist being frozen.

see also:

OpenLookBaseFrame

OpenLookNumeric

Subclass of **ClassControl,ClassBag**
Source file: **OLnumctrl.ps**

This class can be directly instantiated

An Open Look numeric control is a ClassBag that contains an OpenLook-TextControl plus a pair of buttons that increment and decrement the numeric value displayed in the text control. The numeric value may be either integer or real. Many methods that pertain to the text control, such as **/settextparams,** are implemented also in OpenLookNumeric, which simply passes the messages to the underlying text control.

Direct Methods

`/callnotify`

– **/callnotify** –
Call the client's notifyproc, and remember the current value for use by **/checknotify.**

`/checknotify`

`object|null` **/checknotify** –
Call the client's notifyproc only if the current value differs from the value as of the last call to **/callnotify.** This method is called automatically whenever the text control calls *its* notifyproc, i.e., when RETURN is typed or the control loses the input focus after the text has been changed. (Note that it is possible for the text to change without changing the numeric value, e.g., by adding zeroes at the front.)

`/cleartarget`

`object|null` **/cleartarget** –
Selectively clear the target used by the **/sendtarget** method. This method is called automatically when the numeric control is destroyed. See ClassTarget for more about targets and their uses.

`/decrement`

– **/decrement** –
Decrement the value by one unit. The unit defaults to 1, but can be changed by the **/setincrement** method. The decrement button sends a **/decrement** message.

`/demo`

– **/demo** `instance`
Create a sample numeric control whose parent is the framebuffer.

`/destroy`

– **/destroy** –
Destroy this control. This recursively destroys the text control and buttons.

`/disable`

– **/disable** –
Cause the control to stop responding to user actions. This recursively disables the text control and buttons.

`/enable`	– **/enable** – Cause the control to respond to user actions. This recursively enables the text control and buttons.
`/enabled?`	– **/enabled?** `bool` Return **true** if the control is currently enabled.
`/increment`	– **/increment** – Increment the value by one unit. The unit defaults to 1, but can be changed by the **/setincrement** method. The increment button sends a **/increment** message.
`/location`	– **/location** `x y` Return the location of the origin (normally the lower left corner) of the control relative to the CTM.
`/minsize`	– **/minsize** `minwidth minheight` Return the minimum size for this control. In addition to leaving room for a specified number of characters and text scroll buttons (see *ClassTextControl* and *OpenLookTextControl*), this method leaves room for the increment or decrement buttons.
`/move`	`x y` **/move** – Move the origin (lower left) of the control to the specified location in the coordinates of the CTM.
`/new`	`notifyproc parentcanvas` **/new** `instance` Return a new numeric control instance with the specified callback proc and parent canvas. The control initially contains no text; if asked for its **/value** it will return the **/DefaultValue** (q.v.).
`/newinit`	`notifyproc` **/newinit** – Create the enclosed text control and buttons, then store the notifyproc using **/newinit super send.**

`/notifiedvalue`

– **/notifiedvalue** `any`
Return the value the control had the last time **/callnotify** was sent.

`/notifyproc`

– **/notifyproc** `proc`
Return the current callback proc without invoking it.

`/preferredsize`

– **/preferredsize** `preferredwidth preferredheight`
Numeric controls do not presume any preferred size beyond the minimum required. Thus they use the default **/preferredsize** method, which simply calls **/minsize.**

`/reshape`

`x y w h` **/reshape** –
Reshape the control to fit the bounding box specified by the arguments, in the coordinates of the CTM.

`/sendtarget`

`args /method` **/sendtarget** `results`
This method can be invoked within the notifyproc in order to send a message to some other instance. Many clients will not need to use this indirection. See ClassTarget for more details.

`/setcolors`

`strokecolor fillcolor textcolor` **/setcolors** –
Set the colors for painting the text and its background. **Null** as an argument means do not change that value. The *strokecolor* is used for painting the buttons. After calling **/setcolors,** clients must call **/invalidate** and then **/paint** to cause the text control to repaint using the new colors.

`/setdisplaychars`

`n` **/setdisplaychars** –
Set the minimum number of characters that the control should be able to display. This message is handed off to the underlying text control, which uses it to determine **/minsize.**

`/setincrement`

`increment` **/setincrement** –
Set the amount by which the value is changed by **/increment** and **/decrement**. A negative value will cause **/increment** to decrease the value and **/decrement** to increase it. Setting the increment to zero causes the **increment/decrement** buttons to be removed from the control. The current increment can be examined via the **/Increment** instance variable.

`/setmax`

`max` **/setmax** –
Set the maximum value permitted for the control. Any attempt to set a value greater than the maximum (by typing a larger number, or by the **/increment** method) sets it instead to the maximum. When the value is equal to the maximum, the increment button is disabled. The initial maximum value is 32767.

`/setmin`

`min` **/setmin** –
Set the minimum value permitted for the control. Any attempt to set a value less than the minimum (by typing a smaller number, or by the **/decrement** method) sets it instead to the minimum. When the value is equal to the maximum, the decrement button is disabled. The initial minimum value is -32767.

`/setnotifyproc`

`proc` **/setnotifyproc** –
Store the proc as the new callback to be used by **/callnotify,** in place of the one provided to **/new.**

`/setrange`

`min max` **/setrange** –
Set both the minimum and maximum values for the control, as described for **/setmin** and **/setmax.**

`/settarget`

`object` **/settarget** –
Set the target used by the **/sendtarget** method. See ClassTarget for more about targets and their uses.

/settextparams	**family pointsize encoding /settextparams** – Set the text parameters that determine the font for the control. This message is handed off to the underlying text control, as well as being applied to the numeric control's own canvas (since it affects the size of the increment or decrement buttons).
/setvalue	n **/setvalue** – Replace the current value with the given number, subject to min/max constraints, and change the text in the text control accordingly.
/size	– **/size** w h Return the width and height of the control in the CTM.
/target	– **/target** object Return the current target (if any) used by **/sendtarget**. See ClassTarget for more details.
/value	– **/value** num Return the current value of the control, obtained by interpreting the text control's value as a PostScript token. If the resulting object is neither **/integertype** nor **/realtype, /value** instead returns the **/DefaultValue** for the control.
/ButtonNotify	buttoninstance **/ButtonNotify** – This is the callback used for the increment/decrement buttons. The default behavior is to send **/value** to the buttoninstance to determine which button has been pressed, and then send either **/increment** or **/decrement,** as appropriate, to the numeric control.
/ButtonY	– **/ButtonY** y Return the y-coordinate for positioning the increment/decrement buttons. By default they are placed so as to align with the baseline of the text.

`/CallBack`

`obj` **/CallBack** –

This is the notifyproc for the underlying text control. It is called when the text control receives a RETURN keystroke or when it loses the input focus, provided the text string has been changed. The default behavior is to normalise the value by applying the min/max constraints and removing extraneous zeroes, then to send **/checknotify** which invokes the numeric control's own callback if the value has changed.

`/DefaultValue`

/DefaultValue

Return a default value to be used if the text string is unreasonable. This message is sent by **/value** and **/CallBack** if the string does not represent an integer or real value. (Thus, in particular, an empty string will yield **/DefaultValue** rather than zero.) The default behavior is to return the value halfway between the minimum and maximum values, truncated to an integer. Note that the default min/max values yield a **/DefaultValue** of zero.

Class Variables

`/Gap`

This is the amount of space to leave between the text control and the increment/decrement buttons. The default is 2.

see also:

ClassControl, ClassBag

OpenLookPane

Subclass of **ClassContainer**
Source file: **OLpane.ps**

This class should be subclassed, rather than directly instantiated

An OpenLookPane is a container with scrollbars and a client canvas. The scrollbars (none, one, or two) are not automatically connected to the canvas; this should be achieved via subclassing.

see also:

ClassContainer, OpenLookVerticalScrollbar

OpenLookPropertyFrame

Subclass of **OpenLookFrame,ClassPropertyFrame**
Source file: **OLframe.ps**

This class can be directly instantiated.

This class is OpenLookBaseFrame with the following differences; property frames:

- have a pin
- do not have a footer
- cannot be closed into an icon (have no "iconic" state)
- cannot be quit from (pressing SELECT over the close box or selecting "Dismiss" from their menu simply unmaps them from the screen.

see also:

OpenLookBaseFrame

OpenLookTextControl

Subclass of **ClassControl**
Source file: **txtctrl.ps**

This class can be directly instantiated

An Open Look text control augments the basic text control with Open Look features such as scroll buttons. (The Open Look selection features are obtained indirectly from ClassTextControl, which hooks into a UI-independent mechanism).

Direct Methods

`/callnotify`

– **/callnotify** –
Call the client's notifyproc, and also make a copy of the string. The copy will not be affected by subsequent user keystrokes, so **/checknotify** will be able to determine whether the value has changed.

`/checknotify`

`object|null` **/checknotify** –
Call the client's notifyproc only if the current text differs from the text as of the last call to **/callnotify.** This method is called automatically when the text control loses the input focus, or when the user types the RETURN key.

`/cleartarget`

`object|null` **/cleartarget** –
Selectively clear the target used by the **/sendtarget** method. This method is called automatically when the text control is destroyed. See ClassTarget for more about targets and their uses.

`/delchar`

n **/delchar** –
Delete abs(**n**) characters from the text, starting at the current caret position. If **n** is negative, delete characters to the left of the caret, otherwise to the right. After the deletion, the text is scrolled if necessary to bring the caret into the visible region.

/delspan

`left rightplusone` **/delspan** –
Delete a span of characters from the text, independent of where the caret is. If the caret and/or a selection is within the deleted region, it is adjusted accordingly. Character positions start with zero; thus **left** is the number of characters before the first to be deleted, and the number of characters deleted is **rightplusone** minus **left**. After the deletion, the text is scrolled if necessary to bring the caret into the visible region.

/delword

– **/delword** –
Delete characters backward from the caret position until (a) at least one alphanumeric has been deleted and (b) the character ahead of the caret is not alphanumeric. The method **/AlphaNumeric?** is used to determine whether a character can be part of a word.

/demo

– **/demo** `instance`
Create a sample text control whose parent is the framebuffer.

/destroy

– **/destroy** –
Destroy this control. In addition to clearing the target and removing the canvas from the canvas tree, etc., this method disables the caret, in case for example the caret needs to stop a forked process used to implement blinking. (The default ClassCaret does not.) The **/destroy** method also clears some cached references to the text control's scroll buttons, so that they can be destroyed as well.

/disable

– **/disable** –
Cause the control to stop responding to user actions. The control repaints to display its disabled status, and will no longer accept the input focus.

/enable

– **/enable** –
Cause the control to respond to user actions,

provided that it is not a read-only text control. The control repaints to display its enabled status, and will become the input focus if the user clicks the mouse over it.

`/enabled?`

– **/enabled?** `bool`
Return **true** if the control is currently enabled, even if it is read-only.

`/fitcaret`

– **/fitcaret** –
Call **/FitCaret**, which scrolls the text left or right to bring the caret into the visible region.

`/fontoffset`

– **/fontoffset** `int`
Compute the vertical offset for drawing characters without cutting them off at the bottom of the canvas, and also leaving enough room for the caret. The value is 'promoted', i.e., the first time **/fontoffset** is called it stores the resulting value in the instance under the name **/fontoffset**, so that subsequent calls are much faster. The cached value is discarded (unpromoted) if the canvas becomes invalid, e.g. if **/settextparams** is called.

`/inserttext`

`char|string` **/inserttext** –
Insert a character or a string into the text at the current caret position. The caret is moved to the end of the inserted text, and the text is scrolled if necessary to bring the caret into the visible region. If a pending-delete primary selection spans the caret, the selected text is deleted before adding the new text. Any other existing selections are adjusted appropriately.

`/invisiblecaret`

– **/invisiblecaret** –
Make the caret invisible, usually in preparation for other operations that will cause painting. This method is not normally used at all, since the methods that cause painting (such as /insert-text, **/delchar**, etc.) all call **/InvisibleCaret**, which

assumes that the text control is the current canvas.

/location

– **/location** x y
Return the location of the origin (normally the lower left corner) of the text control relative to the CTM.

/minsize

– **/minsize** minwidth minheight
Return the minimum size for this control. In addition to leaving room for a specified number of characters (see ClassTextControl), this method adds to the minimum width to leave room for the two scroll buttons.

/move

x y **/move** –
Move the origin (lower left) of the text control to the specified location in the coordinates of the CTM.

/new

notifyproc parent **/new** instance
Return a new text control instance with the specified callback procedure and parent canvas. The control initially contains no text, but does contain a caret.

/newinit

callback **/newinit** –
This method does **/newinit super send** to store the callback procedure,
then performs other initialisation specific to text controls.

/notifiedvalue

– **/notifiedvalue** string
Return the value the control had the last time **/callnotify** was sent.

/notifyproc

– **/notifyproc** proc
Return the current callback proc without invoking it.

/painttext

n **/painttext** –
Paint the text, starting with the character at position nn. If character nn is off the left edge of the canvas, painting starts at the edge of the

canvas. If n < 0, all the text is painted and the remainder of the canvas is cleared; otherwise the canvas is left unchanged beyond the end of the text. The scroll arrows, if present, are restored after the text is painted. This method is intended mainly for use from other methods; clients should generally send **/paint** to paint the control. **/painttext** operates by setting the current canvas and font, then calling the internal methods **/InvisibleCaret, /PaintText,** and **/VisibleCaret.**

`/preferredsize`

– **/preferredsize** `width height`
Text controls do not presume any preferred size beyond the minimum required to display a certain number of characters. Thus they use the default **/preferredsize** method, which simply calls **/minsize.**

`/removefocus`

`event` **/removefocus** –
Handle the input focus being moved to another canvas. This method disables the caret and calls **/checknotify,** which calls the client's notifyproc if the text has been changed.

`/reshape`

`x y w h` **/reshape** –
Reshape the control to fit the bounding box specified by the arguments, in the coordinates of the CTM.

`/restorefocus`

– **/restorefocus** –
Activate the caret if the control is enabled. This method is called automatically when the text control is given the input focus.

`/scroll`

n **/scroll** –
Scroll the text to make a different portion visible within the limits of canvas. The left edge of the canvas always marks the beginning of a character; scrolling changes the character at this location. A positive number nn scrolls to the right; a negative number scrolls to the left.

/sendtarget

args /method /sendtarget results
This method can be invoked within the notifyproc in order to send a message to some other instance. Many clients will not need to use this indirection. See ClassTarget for more details.

/setcolors

strokecolor fillcolor textcolor /setcolors –
Set the colors for painting the text and its background (fillcolor). Null as an argument means do not change that value. The control swaps the fillcolor and textcolor to highlight selections. The strokecolor is not used used for painting the scroll buttons; the underlining is done using the textcolor so that it can be included in the highlighting.

/setdisplaychars

n /setdisplaychars –
Set the minimum number of characters that the control should be able to display. This number is used in computing **/minsize** for the control. The number can be retrieved via the **/Display-Chars** variable.

/setnotifyproc

proc /setnotifyproc –
Store the proc as the new callback to be used by **/callnotify**, in place of the one provided to **/new**.

/setposition

event /setposition –
n /setposition –
Move the caret to the specified position. Position 0 puts the caret to the left of the first character. If an event is given, the coordinates in the event are resolved to the nearest character boundary. The current caret location can be obtained from the variable **/Left**. Once the caret is placed, the text is scrolled if necessary to bring the caret into the visible region.

/setreadonly

bool /setreadonly –
Make the text read-only or not, depending on the bool. If the text becomes read-only, the

canvas is removed from the list of potential input foci; if it becomes writeable it is added back. A read-only text control behaves much the same as a disabled one (see **/disable**), the main difference being that a disabled control will typically be painted light gray, whereas a read-only one will paint normally.

`/settarget`

`object` **/settarget** –
Set the target used by the **/sendtarget** method. See ClassTarget for more about targets and their uses.

`/settextparams`

`family pointsize encoding` **/settextparams**
Set the text parameters that determine the font for the control. If any of the arguments is null, that parameter is not changed. The control then marks itself as invalid (i.e., calls **/invalidate**) so that various cached values such as **/fontoffset** and the font itself will be recomputed.

`/setvalue`

`string` **/setvalue** –
Replace the contents of the text control with the given string. If the string differs from the old contents, the control is repainted.

`/size`

– **/size** `w h`
Return the width and height of the control in the CTM.

`/starttext`

– **/starttext** –
This method enables the caret. It is called from **/restorefocus.**

`/stoptext`

– **/stoptext** –
This method disables the caret. It is called from **/removefocus** and **/destroy.** Clients may also need to send **/stoptext** to newly created text controls to disable their initial carets.

`/strlen`

– **/strlen** `n`
Return the number of characters of text currently stored in the control. This method is

equivalent to, but more efficient than, **/value** *length.*

/target

`-` **/target** `object`
Return the current target (if any) used by **/sendtarget.** See ClassTarget for more details.

/value

`-` **/value** `string`
Return the contents of the control as a PostScript string. If the string is empty, a zero-length string is returned, *not* **null.**

/visiblecaret

`-` **/visiblecaret** `-`
Make the caret visible, usually after other painting has finished. This method is not normally used at all, since the methods that cause painting (such as **/inserttext, /delchar,** etc.) all call **/VisibleCaret,** which assumes that the text control is the current canvas.

Subclass Methods

/AlphaNumeric?

`n` **/AlphaNumeric?** `bool`
Determine whether the character at the specified index is part of a word. This method is called by **/delword,** and also when making word-level selections. The default method texts for letters, digits, and underscore. See **/AlphaNumericTable.**

/CaretPosition

`-` **/CaretPosition** `x`
Return the x-coordinate of the caret relative to the left edge of the canvas.

/DeHighlight

`start end` **/DeHighlight** `last`
Paint the characters in positions start*start* through end*end*-1, and return the index of the next character to be painted. The characters are painted using the canvas's **/FillColor** as background and **/TextColor** for the text and underline. This method is called by **/PaintText** and in turn calls **/PaintNText.**

`/EOL`

`- ` **/EOL** ` -`
This method is called when either a RETURN (^M) or NEWLINE (^J) character is typed into the control. It calls **/checknotify.**

`/FitCaret`

`/method` **/FitCaret** `bool`
Scroll the text so as to bring the current caret position into the region between the left and right edges of the canvas. Return **true** if any scrolling was done, so the caller will know to repaint the text. (**/FitCaret** does not itself do any painting.) The method-name parameter is ignored; OpenLookTextControls scroll after all relevant operations.

`/Highlight`

`start end` **/Highlight** `last`
This is the same as **/DeHighlight** except the background and foreground colors are reversed. It is called by **/PaintText** for painting characters within the primary selection.

`/InterestingRank`

`rank` **/InterestingRank** `bool`
Return **true** if the selection rank is one that the text control knows how to deal with. By default the only interesting ranks are **/PrimarySelection** and **/SecondarySelection.** (**/ShelfSelection** is handled by the global UI mechanism.) For each interesting rank there is also a correspondingly named instance variable, used for maintaining information about that selection. (These variables are of no interest to clients or subclassers, except insofar as subclassers must be careful not to use those variable names for other information.)

`/PaintText`

`n` **/PaintText** `-`
This method performs most of the work of **/painttext**. It can assume that the text control's canvas is the current canvas, and the control's font is the current font. OpenLookTextControl overrides **/PaintText** to include repainting the scroll buttons after the text is drawn.

`/PaintNText`

textcolor backcolor start end /PaintNText last

This method is called by **/Highlight** and **/DeHighlight** to paint portions of the text using the given colors for the text and background. The characters painted are those in positions *start* through *end*-1. This range may be empty. **/PaintNText** returns the index of the next character to be painted, the larger of *end* and *start*. This method assumes that the text control is the current canvas and the control's font is the current font. OpenLookTextControl overrides **/PaintNText** to underline the text (using the supplied *textcolor*).

`/TextBegin`

– /TextBegin –

Save the graphics context, then set the text control's canvas and font as the current canvas and font. This method is called at the beginning of several other methods to establish the context for painting and other CTM-based operations.

`/TextEnd`

– /TextEnd –

Restore the graphics context saved by **/TextBegin.**

Class Variables

`/AlphaNumericTable`

This is a dictionary whose keys are single characters (integers). The associated values are ignored. A character appears as a key if and only if that character is considered part of a word. This dictionary is used by **/AlphaNumeric?**, and can be overridden by subclassers to change the definition of word-selection and **/delword.** See *ClassTextControl* for a warning regarding subclassing this variable.

`/DisplayChars`

The minimum number of characters that the control should be able to display. The default

value is 5, unless overridden by subclassing or by **/setdisplaychars.**

`/Left` — The number of characters to the left of the caret; hence, the current caret position.

`/ReadOnly?` — **True** if the control is read-only. This value defaults to **false**, but can be overridden by subclassers or **/setreadonly.**

see also:

ClassTextControl

OpenLookVerticalScrollbar

Subclass of **OpenLookHorizontalScrollbar**
Source file: **OLsbar.ps**

This class can be directly instantiated

An OpenLookVerticalScrollbar is an analog control that allows the user to scroll through a large document, for example. It is the default vertical scrollbar for an OPEN LOOK pane. This entry also covers OpenLookHorizontalScrollbar. The only differences between the two are:

- OpenLookVerticalScrollbar has an origin in the upper left corner, whereas OpenLookHorizontalScrollbar uses the standard lower left origin.
- The menu associated with OpenLookVerticalScrollbar has "Here to top", whereas OpenLookHorizontalScrollbar has "Here to left"

Direct Methods

`/abbreviated?` — -- **/abbreviated?** `bool`
Returns **true** if the scrollbar is displayed in its abbreviated form: with the two arrow controls, but no drag box.

`/destroy` — -- **/destroy** --
This destroys the scrollbar menu, and then destroys the scrollbar itself.

`/disable`
: `--` **/disable** `--`
This disables the scrollbar. The scrollbar will be repainted to show that it is disabled, and the mouse clicks will have no affect.

`/enable`
: `--` **/enable** `--`
This is the opposite of disable. The scrollbar is repainted to indicate that it is enabled, and mouse clicks will be accepted again.

`/exchlastvalue`
: `--` **/exchlastvalue** `--`
This exchanges the current scrollbar value with the previous value.

`/heretostart`
: `event` **/heretostart** `--`
`x y` **/heretostart** `--`
Increments the scrollbar value by part of a view. /heretostart expects an x/y pair or an event (which it then turns into an x/y pair). The new value of the scrollbar will be such that the part of the document at the x/y position will be displayed at the top of the view. This is how HereToTop is implemented.

`/menu`
: `--` **/menu** `menu`
Returns the scrollbar menu.

`/minsize`
: `--` **/minsize** `minwidth minheight`
Returns the size of the scrollbar in its abbreviated form.

`/new`
: `notifyproc parent` **/new** `scrollbar`
Creates a scrollbar with the specified notify proc. The parent canvas is consumed by /newobject, /notifyproc is consumed by /newinit.

`/newinit`
: `notifyproc` **/newinit** `--`
This method is called by /new to initialize the instance. The notifyproc is consumed by ClassControl's /newinit. This also creates the scrollbar's anchor and elevator controls.

`/preferredsize`

`--` **/preferredsize** `preferredwidth preferredheight`

Often preferredsize is the same as minsize, but in the case of OpenLook scrollbars, /preferredsize is the size to fit the entire elevator. /minsize is the size of an abbreviated scrollbar, and /preferredsize is the next size up from that.

`/setdelta`

`name value` **/setdelta** `--`

The scrollbar deltas are named /Line, /Page and /Document, and they specify how much to increase or decrease the value of the scrollbar when the user selects a particular type of scrolling. Typical values for these deltas might be specified in units of lines, and so /Line would be set to 1, /Page would be the number of lines visible in a page, and /Document would be the number of lines in the document.

`/setmenu`

`menu` **/setmenu** `--`

Set the menu for the scrollbar. You do not normally need to call this method as OpenLook scrollbars come with a standard menu.

`/setrange`

`min max` **/setrange** `--`

Set the range of the scrollbar.

`/starttohere`

`event` **/starttohere** `--`
`x y` **/starttohere** `--`

Decrements the scrollbar value by part of a view. /starttohere expects an x/y pair or an event (which it then turns into an x/y pair). The new value of the scrollbar will be such that the part of the document at the top of the view will now be displayed at the specified x/y position. This is how TopToHere is implemented.

Subclass Methods

/ClientDown

event /ClientDown --
This method is called when the user mouses down over the scrollbar. This saves away the current value of the scrollbar, and then processes the mouse event.

/ClientUp

event /ClientUp --
This method is called when the user lets go of the mouse button that caused **/ClientDown** to be called.

/MaxHit

anchor-control /MaxHit --
Called when the high anchor button is hit.

/MinHit

anchor-control /MinHit --
Called when the low anchor button is hit.

/PaintCable

value /PaintCable --
Called to paint the scrollbar's cable and proportion indicator.

/PaintCanvas

-- /PaintCanvas --
Called to paint the entire scrollbar.

/PaintValue

value /PaintValue --
Paints the scrollbar with specified value.

OpenLookVerticalSlider

Subclass of **OpenLookHorizontalSlider**
Source file: **OLslider.ps**

This class can be directly instantiated

This slider is a horizontal OPEN LOOK slideattr rotated 90°; it has the same interface as OpenLookVerticalSlider.

see also:
OpenLookHorizontalSlider

OpenLookXSetting

Subclass of **RowColumnLayout,ClassSelectionList**
Source file: **OLxset.ps**

This class can be directly instantiated.

This class implements an Open Look exclusive setting group. The group displays multiple setting items on a single canvas. The group acts like a control in that it has a single value. Each item has an associated notify procedure that is called when the item is selected.

Direct Methods

`/border` — `-` **/border** `number`
Return the size of the border around the entire group of settings.

`/change` — `location thing|graphic genproc|sublist|null proc|null` **/change** `-`
Change the setting item at the specified location. The parameters, other than location, are the same as for **/new.**

`/cleartarget` — `object|null` **/cleartarget** `-`
Clear the control's target object. If an object is supplied as a parameter, the target will be cleared only if the current target matches the parameter. See *ClassTarget* for more about targets and their uses.

`/clientcount` — `-` **/clientcount** `n`
Return the number of items in the setting.

`/clientlist` — `-` **/clientlist** `array`
Return an array of dictionaries, one per item in the control group.

`/delete`	`location` **/delete** – Delete the specified setting item from the group.
`/demo`	– **/demo** `instance` Create a demonstration exclusive setting on the framebuffer.
`/destroy`	– **/destroy** – Destroy the exclusive setting control.
`/disableitem`	`index` **/disableitem** – Disable the item at the specified location.
`/doaction`	– **/doaction** – Execute the notify proc associated with the currently selected setting item. i.e. the item whose index matches the setting's current value.
`/enableitem`	`index` **/enableitem** – Enable the item at the specified location.
`/gaps`	– **/gaps** `horizontalgap verticalgap` Return the amount of space between the setting items. By default, and by the OPEN LOOK specification, the values are both 0.
`/graphic`	`location` **/graphic** `graphic` Return the graphic for the specified setting item.
`/insert`	`location thing|graphic genproc|sublist|null proc|null` **/insert** – Insert a new setting item at the specified position in the group. The parameters, other than location, are the same as for **/new**.
`/invalidate`	– **/invalidate** – Invalidate the setting and all of its item graphics.
`/itemcount`	– **/itemcount** `integer` Return the number of items in the setting group.
`/itemenabled?`	`location` **/itemenabled?** `bool` Return **true** if the specified item is enabled.

/layout	– **/layout** – Lay out the setting items. This sizes and places the items according to the layout style for the setting.
/layoutstyle	– **/layoutstyle** `RowMajor? Rows|null Columns|null` Return the layout style for the menu. See class *RowColumnLayout*.
/location	– **/location** `x y` Return the location of the setting canvas.
/maxlocation	– **/maxlocation** `location` The location of the last item in the control. Item locations range from 0 to **/maxlocation. /maxlocation** is one less than **/itemcount.**
/minsize	– **/minsize** `minwidth minheight` Return the minimum size of the setting.
/move	`x y` **/move** – Move the setting's canvas to the specified location, relative to the CTM.

/new	`[thing|graphic proc|null ...]`
	`canvas` **/new** `instance`
	Create an exclusive setting group. Refer to the introductory section *OPEN LOOK Settings* for the details of the arguments.

/notifyproc	`location` **/notifyproc** `procedure|null` Return the notify proc for the item at the specified location.
/preferredsize	– **/preferredsize** `width height` Return the preferred size of the setting group.
/reshape	`x y w h` **/reshape** – Reshape the setting group's canvas. Invalidates the setting group.
/searchgraphic	`graphic` **/searchgraphic** `index true` `graphic` **/searchgraphic** `false`

	`graphic startloc` **/searchgraphic** `index true`
	`graphic startloc` **/searchgraphic** `false`
	Search for the setting item having the given graphic. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/searchthing	`thing` **/searchthing** `index true`
	`thing` **/searchthing** `false`
	`thing startloc` **/searchthing** `index true`
	`thing startloc` **/searchthing** `false`
	Search for the setting item having the given thing in its graphic. If a starting location is given, begin the search at that item. The search ends with the last menu item and does not wrap around.

/sendtarget	`args /method` **/sendtarget** `results` Send a message to the target object. Typically used in the control's notify proc. See ClassTarget for more details.
/setborder	`number` **/setborder** – Set the size of the border around the entire group of settings.
/setgaps	`horizontalgap|null` `verticalgap|null` **/setgaps** – Change the amount of space between the setting items. A null parameter leaves that value unchanged.
/setgraphic	`location thing|graphic` **/setgraphic** – Change the graphic for the specified setting item.
/setlayoutstyle	`RowMajor? Rows|null` `Columns|null` **/setlayoutstyle** –
/setlayoutstyle	`RowMajor? Rows|null` `Columns|null` **/setlayoutstyle** – Change the layout style of the menu.

See *RowColumnLayout*.

`/setnotifyproc`

`location procedure|null` **/setnotifyproc** `-`
Change the notification procedure (callback) associated with a setting item. Note that if you specify a single notify proc when creating the setting group, the notify proc is duplicated for each item. You may therefore change individual item callbacks without affecting others. Also, to change the callbacks for all items, you must call **/setnotifyproc** for each item.

`/settarget`

`object` **/settarget** `-`
Set the target used by the **/sendtarget** method. See *ClassTarget* for more about targets and their uses.

`/setvalue`

`index` **/setvalue** `-`
Change the value of the setting. The item corresponding to the supplied index is selected and the previously selected item, if there was one, is deselected.

`/size`

`--` **/size** `w h`
Validate the setting and return its size in the coordinates of the CTM.

`/target`

`-` **/target** `object`
Return the target object. See *ClassTarget*.

`/value`

`-` **/value** `index`
Return the location of the selected item. The first item is number 0. Returns null if there is no selected item. Note that this can happen if the value is never changed from its initial value or if it is explicitly set to null with **/setvalue.** It can not happen as the result of a user interaction.

`/valuething`

`-` **/valuething** `thing|null`
Return the thing of the graphic of the selected item. Returns null if the **/value** of the setting is null.

Subclass Methods

`/DeHighlight`	`index` **/DeHighlight** –
`/Highlight`	`index` **/Highlight** –

see also:

RowColumnLayout,ClassSelectionList

OpenLookXSettingControl

Subclass of **OpenLookButton**
Source file: **OLxctrl.ps**

This class should be subclassed, rather than instantiated directly.

This class exists only as a superclass to OpenLookCheckBox. Its purpose is to make a control out of a single setting item. .

see also:

OpenLookButton, OpenLookCheckBox

RowColumnBag

Subclass of **RowColumnLayout,ClassBag**
Source file: **bagutils.ps**

This class may be directly instantiated.

A RowColumnBag is a general purpose bag in which the clients are layed out on an equally spaced matrix. The interface for controlling this matrix is the same as that offered in class RowColumnLayout, one of the ancestors of this class. The interface is also similar to that used in OpenLookMenu for laying out menu items.

Direct Methods

/addclient

name|null client /addclient –
name|null [client_class args client_class] /addclient –
Add a client to the bag. The client may be specified by either an instance or by a class to instantiate, with the arguments necessary to do so.

No baggage needs to be specified, since the clients will be layed out as a matrix specified by **/setlayoutstyle**. The clients are stored in a linear array in the order they are inserted into the bag. This array is then mapped onto the matrix that is displayed.

/border

– **/border** int

/clientcount

– **/clientcount** n
Return the number of clients currently in the bag.

/clientlist

– **/clientlist** [client1 client2 ...]
Return an array of clients in the same order as they were inserted into the bag.

/destroy

– **/destroy** –
Destroy the bag and its clients. Refer to the */destroy* method in *ClassBag* for the additional information on the use of this method.

/gaps

– **/gaps** horizontalgap verticalgap
Return the horizontal and vertical gaps between cells in the matrix to be layed out, as specified by **/setgaps**.

/layoutstyle

– **/layoutstyle** RowMajor? Rows|null Columns|null
Return the layout specification as described in class *RowColumnLayout*. The three arguments returned indicate whether row or column order is used, how many rows, and how many columns.

/location

- /location x y
Return the location of the origin of the bag in the coordinates of the CTM.

/minsize

/minsize minwidth minheight
Compute the minimum acceptable size for this bag, as if all clients were as big as the largest minimum size of any client. A well behaved application will respect this size when reshaping the bag in response to user mouse actions.

/move

x y /move -
Move the origin of the canvas to the specified location in the coordinates of the CTM.

/new

parentcanvas /new instance
Create an instance of class RowColumnBag.

/preferredsize

- /preferredsize preferredwidth preferredheight
Calculate the "ideal" size of the bag, which defaults to the minimum size. Well behaved applications should respect this size when initially displayed the bag.

/removeclient

client|name|n /removeclient oldclient true
client|name|n /removeclient false
Remove the client given, named or indexed in the argument. The method returns **true** and the client object if the client is found, otherwise it returns **false**. Refer to the **/removeclient** method in *ClassBag* for the additional information on the use of this method.

/reshape

x y w h /reshape -
Reshape the bag to the dimensions given and invalidate it. This will later result in the bag being layed out as the first step in painting it.

/sendclient

<args> /method /name /sendclient results
Send the given method with arguments to the named client.

/setborder

`number /setborder -`
Set the space used at the border of the matrix, and invalidate to mark the matrix as needing layout.

/setgaps

`horizontalgap|null verticalgap|null /setgaps -`
Set the horizontal and vertical spacing between cells in the matrix, and invalidate to mark the matrix as needing layout. If either argument is null, that spacing is not changed.

/setlayoutstyle

`RowMajor? Rows|null Columns|null /setlayoutstyle -`
Set the layout parameters of the matrix, and invalidate to mark it as needing layout.

/size

`- /size w h`
Return the width and height of the canvas in CTM.

see also:

RowColumnLayout,ClassBag

RowColumnLayout

Subclass of **Object**
Source file: **OLutil.ps**

This class should be mixed into another subclass rather than directly instantiated.

This class is a mix-in to perform grid-style layout and point location on a canvas. It has a similar interface and behavior to the layout code found in ClassMenu.

The basic idea behind laying out rows and columns is to specify the size of a 2 dimensional matrix and how to fill it in from a 1 dimensional array of elements. The argument listed below as `RowMajor?` is a boolean specifying how to fill in the matrix:

true = Row major order, fill rows first
false = Column major order, fill columns first

The size of the matrix is specified by arguments listed below as `Rows|null` and `Columns|null`.

- both arguments are specified numerically,
  the layout has that number of rows and columns, and if there are more items than will fit into the specified matrix, those items are not displayed.
- both are null,
  use the default layout specification, as given by **/setlayoutstyle**.
- one is specified numerically and one is **null**,
  then the matrix has the specified number of rows or columns and whatever number of columns or rows are needed to display all the items.

Thus, a 7 item array with 3 rows and null columns is displayed as follows, depending on whether row major or column major order was specified:

Row Major			Column Major		
0	1	2	0	3	6
3	4	5	1	4	
6			2	5	

Direct Methods

`/border`

– **/border** `number`
Return the size of the borders around the matrix.

`/cellcount`

– **/cellcount** `columns rows`
Return the number of columns and rows to be used in laying out the matrix.

`/gaps`

– **/gaps** `horizontalgap verticalgap`
Return the horizontal and vertical gaps between cells in the matrix to be layed out.

`/invalidate`

– **/invalidate** –
Remove previously stored values for heights and widths and invalidate the matrix to mark it as needing layout. The values discarded are: /ArrayHeight, /ArrayWidth, /MinCellHeight, /MinCellWidth, /CellHeight, /CellWidth, /CellCols, /CellRows.

/layoutstyle

- /layoutstyle RowMajor? Rows|null Columns|null

Return the layout specification, as described above: row or column order, how many rows and how many columns.

/setborder

number /setborder -

Set the space used at the border of the matrix, and invalidate to mark the matrix as needing layout.

/setgaps

horizontalgap|null verticalgap|null /setgaps -

Set the horizontal and vertical spacing between cells in the matrix, and invalidate to mark the matrix as needing layout. If either argument is **null**, that spacing is not changed.

/setlayoutstyle

RowMajor? Rows|null Columns|null /setlayoutstyle -

Set the layout parameters of the matrix, and invalidate to mark it as needing layout.

Subclass Methods

/ArrayHeight

- /ArrayHeight arrayheight

Return the height of the entire matrix, with gaps and borders.

/ArrayWidth

- /ArrayWidth arraywidth

Return the width of the entire matrix, with gaps and borders.

/CellCols

- /CellCols cols

Return the number of columns to be used in laying out the matrix. This number was either supplied as an argument or it is calculated from the specified layout and the number of rows.

/CellHeight

- /CellHeight cellheight

Return the height of a cell in the matrix. This is the maximum height of any item plus the vertical cell spacing (**CellVertGap**).

`/CellRows`

– **/CellRows** `rows`
Return the number of rows to be used in laying out the matrix. This number was either supplied as an argument or it is calculated from the specified layout and the number of columns.

`/CellWidth`

– **/CellWidth** `cellwidth`
Return the width of a cell in the matrix. This is the maximum width of any item plus the horizontal cell spacing (**CellHorzGap**).

`/Layout`

– **/Layout** –
This method calls /Rowcolumnlayout to perform the actual layout of the matrix.

`/MinSize`

– **/MinSize** `minw minh`
Returns the minimum width and height for the matrix. This is determined by the largest minimum size of all objects being layed out.

`/XYToValue`

`x y` **/XYToValue** `index|null`
This method converts a location on the canvas into the index of the matrix item that contains the location. The method returns **null** if the location is not in the canvas area occupied by the matrix.

Class Variables

`/Border` — This variable stores the size of the border.

`/CellHorzGap` — This variable stores the horizontal gap (distance) between cells in the layout.

`/CellVertGap` — This variable stores the vertical distance between cells in the layout.

`/LayoutCols` — This variable stores the number of columns to be used in laying out the matrix; by default the value is 1

`/LayoutRows` — This variable stores the number of rows to be used in laying out the matrix. By default the

value is **null**, which means to use as many rows as necessary.

`/RowMajor?` — This variable stores the boolean specifying row major order (**true**) or column major order (**false**).

see also:

Object

320–720

UNIX
PRESS

A Prentice Hall Title

ISBN 0-13-931858-5