4Sight Programmer's Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-2001-030

4Sight Programmer's Guide

GL/DGL Interfaces
NeWS
Window Manager

Document Version 3.1

Document Number 007-2001-030
(Includes *4Sight Programmer's Guide Update*
Document Number 007-2001-031)

11/90

Technical Publications:

Robert Reimann, Claudia Lohnes, Kevin Walsh

Engineering:

Peter Broadwell, Mark Callow, Dave Ciemiewicz
Kipp Hickman, Allen Leinwand, Rob Myers
Gary Tarolli, Michael Toy, Glen Williams

4Sight Programmer's Guide
Document Version 3.0
Document Number 007-2001-030
(Includes 4Sight Programmer's Guide Update
Document Number 007-2001-031)

Silicon Graphics, Inc.
Mountain View, California

Contents

List of Tables

List of Figures

Introduction

4Sight is an integrated windowing environment for IRIS workstations. With 4Sight, you can combine the best aspects of network-distributed 2-D graphics and high-performance 3-D graphics seamlessly on the IRIS screen.

4Sight supports programs that employ the high-performance, dedicated graphics available from the IRIS Graphics Library, as well as programs written for *NeWS*, and *X11*, two industry-standard networked graphics protocols. 4Sight also supports the Distributed Graphics Library (DGL), a version of the IRIS Graphics Library that allows GL programs to be displayed across a network in a fashion similar to that of NeWS client programs. Multiple windows containing GL, DGL, PostScript or X11 client programs can exist on the screen simultaneously, under the control of a single, uniform window manager. Using PostScript, you can customize and extend the appearance and function of the windows, icons, menus, and controls which make up the window manager's user interface.

1.1 4Sight Design Overview

4Sight manages the screen and input resources (e.g., the mouse and keyboard) of the IRIS workstation. These resources are managed by 4Sight under a *client/server* paradigm. Graphics programs (clients) request screen space in the form of windows from a server that manages the limited screen area on their behalf.

4Sight currently supports four different kinds of graphics clients: GL, DGL, PostScript, and X11. These clients are supported via three distinct servers: the DGL server, the NeWS server, and the X11 server. These servers are integrated within 4Sight to provide a unified set of window services and a seamless appearance on the screen. Each server presents the specific communications interface(s) appropriate for its matching client programs.

Figure 1 shows the relationships between the two 4Sight servers and their clients.

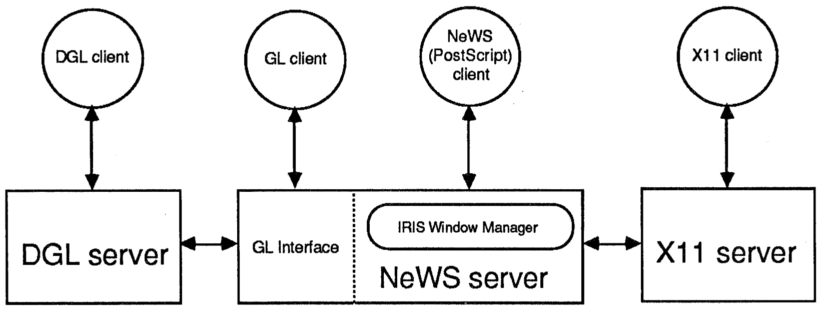


Figure 1-1. 4Sight Interface Architecture

The following two sections briefly describe the three kinds of 4Sight clients, how they relate to their respective servers, and their respective features. Following these is a fourth section describing the IRIS Window Manager.

1.1.1 The GL/DGL Interfaces

Dedicated, high-performance 3-D graphics are supported in 4Sight by the Graphics Library (GL) interface, an extension to the NeWS server. The Graphics Library uses this interface for its windowing functions; use of the GL interface guarantees the highest graphics performance available on the IRIS workstation.

The Distributed Graphics Library (DGL) interface consists of a separate server that communicates both with DGL client programs and with the 4Sight NeWS server. The DGL server allows you to display the graphics of a GL program on a remote machine, via a TCP/IP connection. This gives the IRIS a 3-D networked graphics capability, though at reduced performance due to limitations in communication speed over a network.

For more information on program development using the GL interface and the DGL, see Section 1 of this manual, ''Using the GL/DGL Interface'', and the *Graphics Library Programming Guide.*

1.1.2 NeWS

The Network-extensible Window System (NeWS) server is the central functional component in 4Sight's architecture. NeWS is a distributed, extensible window system developed by Sun Microsystems. In its central organizing role under 4Sight, the NeWS server manages access to the IRIS hardware for event generation and distribution, window manipulation, and display resource arbitration (See ''The IRIS Window Manager'', below). In addition, the NeWS server supports 2-D network-distributed graphics based on the PostScript imaging model. NeWS application programs access these services via their own NeWS client interface, which communicates with the central NeWS server. For more information on NeWS client programs, see Section 2 of this manual, ''Programming in NeWS''.

1.1.3 The X Window System

The X11 server provides an interface for programs written for the X Window System, Version 11R3, developed by an industry consortium at MIT. X11 was designed to provide a distributed, standard window system for the UNIX workstation community. X11 is provided in 4Sight so you can run local or remote X11 clients on your IRIS screen. X11 is provided as an execute-only environment shipped standard with 4Sight. You can order the X11 development package as a software option from Silicon Graphics, Inc.

1.1.4 The IRIS Window Manager

Although 4Sight provides the raw facilities for window management, it does not implement any specific user interface itself. Instead, the IRIS Window Manager is implemented in NeWS; it is a PostScript process running inside the NeWS server. The Window Manager consists of PostScript routines defining the appearance and function of windows, icons, menus, and controls, and the way input events interact with them. Thus, to customize or extend the Window Manager, you need only alter these PostScript functions, or add new ones to suit your needs. For more information, see Section 3 of this manual, "Programming the IRIS Window Manager".

1.2 Important Reading

Section 1 of this manual, "Using the GL/DGL Interface", assumes knowledge of the material covered in the *Graphics Library Programming Guide.*

Sections 2 and 3 of this manual, "Programming in NeWS", and "Programming the IRIS Window Manager", assume knowledge of the material covered in the *PostScript Language Reference Manual.* If you are unfamiliar with PostScript, you should also consider the companion book *PostScript Language Tutorial and Cookbook* to be required reading.

1.3 Style Conventions

This manual uses the following style conventions:

- IRIX™ commands and filenames appear in *italics*, and references to entries in the IRIX documentation are followed by a section number in parentheses. For example, *cc*(1) refers to the *cc* manual entry in Section 1 of the *IRIX User's Reference Manual.*
- In IRIX command line syntax descriptions, square brackets surrounding an argument indicate that it is optional. Variable parameters in IRIX command line syntax descriptions appear in *italics*.
- In text descriptions, command line options are denoted by **bold** font (e.g., the **–l** option to *lp*).
- In text descriptions, IRIS Graphics Library and DGL subroutines appear in `typewriter` font. PostScript (NeWS) commands and operators follow the style conventions adopted by the *PostScript Language Reference Manual.*

- All example programs and code segments appear in `typewriter` font.
- System messages in examples appear in `typewriter` font. Example text that you must type appears in **`bold typewriter`** font.

Section 1:

Using the GL/DGL Interfaces

Document Version 3.1

1. Introduction

The Graphics Library interface (GL interface) to the NeWS server allows you to create programs that use the high-performance 3-D graphics provided by the Graphics Library in combination with the user interface provided by the 4Sight windowing environment.

This part of the *4Sight Programmer's Guide* covers five major topics:

- making GL windows
- making pop-up menus
- controlling several GL windows from a single process
- using the IRIS Font Manager
- using the Distributed Graphics Library (DGL)

There is also an appendix that provides a list of *wsh* (*textport*) escape sequences, the GL routines that issue some of these escape sequences, and an appendix describing performance issues involving the DGL.

The GL interface consists of Graphics Library routines that pass information to 4Sight's NeWS server. This allows the window manager to accommodate the windowing needs of the GL client program. The following chapters list and briefly describe each routine of the GL interface. Because the GL interface supports both C and FORTRAN, the descriptions for all of these routines provide syntax information in both languages.

The DGL interface consists of a separate server that communicates both with DGL client programs and with the 4Sight NeWS server. You can run almost any existing GL program as a DGL client simply by re-linking it with the DGL library. See Chapter 6, ''Using the Distributed Graphics Library'', for complete instruction on its use.

2. Making GL Windows

This chapter explains how to create and manipulate GL windows from an application program. Section 2.1 describes input focus in the IRIS Window Manager. Sections 2.2 through 2.6 describe the Graphics Library routines, which are the building blocks for programming with the GL interface. Section 2.7 gives general hints for writing GL client programs. Sections 2.8 and 2.9 contain sample programs: one that uses single buffer mode and one that uses double buffer mode.

A *GL window* is a window in which an Graphics Library application program draws an image. A *text window* runs a IRIX shell. You create GL windows with the Graphics Library routines described in this chapter.

2.1 Input Focus

The 4Sight window manager uses the ''follow focus'' input focus paradigm. Input focus goes to the window over which the mouse cursor lies.

To direct input to a window that is not underneath the mouse cursor:

1. move the mouse cursor over the window to which you want to direct input
2. press and hold any key (the `Ctrl` or `Shift` keys are good choices)
3. move the mouse out of the window border while still holding the key

Your input remains directed to the original window until you let go of the key; at that point, input goes to whatever lies underneath the cursor.

2.2 Opening and Closing Windows

Use the GL `winopen` routine to open a window. Use the GL `winclose` routine to close a window.

winopen

Use `winopen` to create a graphics window as defined by the current values of the window constraints. This new window becomes the current window. If this is the first time that your program has called `winopen`, the system also initializes the graphics system.

Except for size and location, the system maintains default values for the constraints on a window. You can change these default window constraints if you call the routines `minsize`, `maxsize`, `keepaspect`, `prefsize`, `prefposition`, `stepunit`, `fudge`, `iconsize`, `noborder`, `noport`, `imakebackground`, and `foreground` before you call `winopen`. If a window's size and location (or both) are left unconstrained, the system allows the user to place and size the window.

After opening a window, `winopen` resets all global state attributes (this includes window constraints) to their default system values. (For a listing of the global state attributes and their default values, see the documentation for `greset`.) `winopen` also queues the pseudo devices INPUTCHANGE and REDRAW.

When using the Distributed Graphics Library (DGL), the window identifier also identifies the window's graphics server. A valid window identifier consists of a 32-bit positive integer. The DGL directs all graphics input and output to the current window's server; subsequent Graphics Library subroutines are executed by the window's server.

The *name* argument to `winopen` appears on the window's title bar. When you stow the window to an icon, this text also appears on the window's icon. (Use `wintitle` to change the caption that appears on the title bar Use `icontitle` to change the caption that appears on the icon.) You can also use `winopen` to place specified windows on the desktop, as described in Section 3 of this manual, ''Programming the IRIS Window Manager''.

If you are programming in FORTRAN, `winope` takes an additional argument, `length`, the length of the string supplied as the `name` argument.

The returned value for `winopen` is the graphics window identifier for the window just created. Use this value to identify the graphics window to other graphics functions. If no additional graphics windows are available, this function returns –1.

For information on using `winopen` with the Distributed Graphics Library (DGL), see Chapter 6, ''Using the Distributed Graphics Library''.

```
long winopen(name)
char name[];

integer*4 function winope(name, length)
character*(*) name
integer*4 length
```

winclose

`winclose` removes a window. This routine is only useful for multi-window applications. Exiting a program automatically closes any windows it created. The `gid` parameter is the graphics identifier for the window. (The returned value of the `winopen` function that created the window.)

```
void winclose(gid)
long gid;

subroutine winclo(gid)
integer*4 gid
```

2.3 Setting Window Constraints

You can control the size, location, and shape of graphics windows from a GL client program. Calling `winopen` without first specifying any of these characteristics allows you to open a window of any size or shape anywhere on the screen. But if you want something specific, such as a small square window with a border, you can specify the desired size and shape.

Use the window constraint routines (see Table 2-1) to specify window characteristics. Call these routines before opening a window with `winopen`. (For instructions on changing the characteristics of an existing window, see the description of `winconstraints`.)

When you interactively change a window, the 4Sight server (NeWS) applies the constraints that you specify. Interactively changing a window means sweeping out a window with the mouse, or using pop-up menus to change the size, shape, or position of a window. To change windows noninteractively (that is, from an application program), see Section 2.4.

Use	To specify
`foreground`	Program runs in foreground
`fudge`	Small increase in size
`iconsize`	Size of window's icon
`imakebackground`	Process that draws background
`keepaspect`	Aspect ratio
`maxsize`	Maximum size
`minsize`	Minimum size
`noborder`	No window borders
`noport`	Graphics routines without window
`prefposition`	Size and location
`prefsize`	Size, in pixels
`stepunit`	Sizing increment

Table 2-1. Window Constraint Routines

foreground

`foreground` makes a program run in the foreground. By default, when you first call `winopen` in your program, the 4Sight server runs your program in the background. If you precede your first call to `winopen` with a call to `foreground`, the 4Sight server runs your graphics program as a foreground process. When a program runs in the background, it does not normally receive input from the stdin.

A call to `foreground` applies only to the first call to `winopen`.

```
void foreground()
```

```
subroutine foregr
```

fudge

`fudge` adjusts the size of a graphics window so that a program can draw a border or add a heading. You can use `fudge` and `stepunit` together.

```
void fudge(xfudge, yfudge)
long xfudge, yfudge;
```

```
subroutine fudge(xfudge, yfudge)
integer*4 xfudge, yfudge
```

iconsize

`iconsize` specifies the new size (in pixels) of a window when it is redrawn or "stowed" as an icon. (Call `iconsize` before you call winopen or, if the window already exists, before you call `winconstraints`. See 2.3.1.)

When users stow a window, the 4Sight server sends a REDRAWICONIC event to the graphics queue. Your program can then read the REDRAWICONIC event from the event queue and, if you want, describe the icon to use for the window. If your program gives no special icon description, the icon is the image of whatever the window contained when the user stowed the window.

Note: Programs using `iconsize` must call `reshapeviewport` for a REDRAWICONIC event. This ensures that the image drawn for the icon is appropriately scaled. Such programs must also call `qdevice` for both the WINFREEZE and WINTHAW events (see Section 2.7.4, ‘‘Window Manager Devices’’).

If your program does not call *iconsize*, the window manager handles all the details of freezing the window, drawing the icon, and thawing the window when the user "opens" the icon (see *winicons*(5W)). Your program only needs to call *iconsize* when you want to take control of how the icon is drawn.

```
void iconsize(x,y)
long x, y;

subroutine icons(x,y)
integer*4 x, y
```

imakebackground

imakebackground sets up a process that draws the background of the window manager's screen. Use this routine to create nonstandard (for example, patterned) backgrounds. The process must first draw the background and then read the event queue. The process redraws the background for every REDRAW event in the event queue (see Section 2.7.3). A program that declares itself as imakebackground stops any previous imakebackground program.

```
void imakebackground()

subroutine imakeb
```

keepaspect

keepaspect specifies the aspect ratio (proportions or width-to-height ratio) of the graphics window. You can resize the graphics window, but its aspect ratio stays the same. Use keepaspect(1,1) to ensure that the graphics window is always square. Use keepaspect(4,3) to ensure that the graphics window maintains the ratio four units wide to three units high, no matter how large or small the user resizes the window to be.

```
void keepaspect(x, y)
long x, y;

subroutine keepas(x, y)
integer*4 x, y
```

maxsize

maxsize specifies a maximum size for the graphics window. The default maximum size is 1280 pixels wide (x) by 1024 pixels high (y).

```
void maxsize(x, y)
long x, y;

subroutine maxsiz(x, y)
integer*4 x, y
```

minsize

`minsize` specifies a minimum size for the graphics window. You cannot interactively reshape the graphics window to be smaller than this minimum size. The default minimum size is 80 pixels wide by 40 pixels high.

```
void minsize(x, y)
long x, y;

subroutine minsiz(x, y)
integer*4 x, y
```

noborder

By default, a window has a border around it. To make a new window without borders, call `noborder` before you open the window. To remove the borders from an existing window, use `noborder` with `winconstraints` (see section 2.3.1).

```
void noborder()

subroutine nobord
```

noport

`noport` tells the 4Sight server that the program requires no screen space. You can use this routine in programs that use the GL only to read or write the color map. After a call to `noport`, the next call to `winopen` will not create a window on the screen, but all graphics commands remain enabled. Calling `winconstraints` resets `noport`.

```
void noport()

subroutine noport
```

prefposition

`prefposition` specifies the preferred location and size of the graphics window. The window initially appears on the screen at the preferred location, but you can subsequently drag the window to a different location. You cannot resize a window that you create when prefposition is active.

```
void prefposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

subroutine prefpo(x1, x2, y1, y2)
integer*4 x1, x2, y1, y2
```

prefsize

`prefsize` specifies the size of the graphics window as *x* pixels by *y* pixels. When `prefsize` is in effect, you cannot interactively resize the graphics window. When you open a window with a preferred size, you automatically grey out and deactivate the "Resize" item from the Window (frame) menu. You also automatically hide and deactivate the resize brackets normally displayed in the window corners.

```
void prefsize(x, y)
long x, y;

subroutine prefsi(x, y)
integer*4 x, y
```

stepunit

`stepunit` enforces a "granularity" to changes in the size of a graphics window. All changes to the size of the graphics window are in steps of *xunit* pixels in the horizontal and *yunit* pixels the vertical. See the *Graphics Library Reference Manual* for more information.

```
void stepunit(xunit, yunit)
long xunit, yunit;

subroutine stepun(xunit, yunit)
integer*4 xunit, yunit
```

2.3.1 Setting Constraints for Existing Windows

You can call any of the 12 window constraint routines just before you call `winopen`. The 4Sight server applies these constraints when it opens the window. To respecify constraints for a window that is already open, follow these steps:

1. Call the desired series of window constraint routines (see Table 2-1).
2. Call `winconstraints`.

winconstraints

`winconstraints` restricts the size, location, or shape of a window as specified by the first 11 routines in Table 2-1. `foreground` does not work with `winconstraints`.

```
void winconstraints()

subroutine wincon
```

`winconstraints` performs two functions:

1. It sends any current constraints on the window to the 4Sight server.
2. It resets all constraints to their default values (excluding `foreground`).

The code sample below removes all old constraints and subjects future resizing of the window to a 400x400 pixel minimum, with a square aspect ratio.

```
winconstraints();
minsize(400, 400);
keepaspect(1, 1);
winconstraints();
```

When you call `winconstraints`, the appearance of the window does not immediately change. To put the changes into effect, you must interactively reshape the window with the mouse or from a pop-up menu. `winconstraints` uses the constraints that were set since the last time you created a window or called `winconstraints`.

To remove all previous constraints, call `winconstraints` twice without any new constraints; the first call resets any constraints which may have been set previously, the second call sends the default values of the constraints to the 4Sight server.

```
winconstraints();
winconstraints();
```

So, to open a window that starts out having a 1:1 aspect ratio, which under some condition (such as a menu choice) can not grow larger than 400 pixels by 400 pixels (while maintaining 1:1 aspect ratio), and under a different condition can grow to any size with any aspect ratio, you would write code that looks like this:

```
keepaspect (1, 1);
winopen(name);
...code...
maxsize(400, 400);
keepaspect(1, 1);
winconstraints();
...more code...
winconstraints();
winconstraints();
...remainder of program ...
```

2.4 Changing Windows Noninteractively (from within a Program)

This section describes how to give a program the same control over windows that you have when using pop-up menus. You can move, resize, push, and pop windows from your program. Table 2-2 lists the routines that perform these tasks.

Routine	Task
`winmove`	Move the current graphics window
`winpop`	Pop the current graphics window to front
`winposition`	Move and reshape the current graphics window
`winpush`	Push the current graphics window to back

Table 2-2. Window Control Routines

The 4Sight server performs the operations in Table 2-2 on the current graphics window, which you can set with `winset` (see Section 2.5, Other Window Routines).

winmove

`winmove` moves the lower-left corner (origin) of the current graphics window to screen location *(x, y)*, measured in pixels. The size and shape of the window do not change. The 4Sight server erases the parts of the window over the background and sends REDRAW events to any windows exposed by the move.

```
void winmove(orgx, orgy)
long orgx, orgy;

subroutine winmov
integer*4 orgx, orgy
```

`winposition` and `winmove` act on the current graphics window despite any constraints you have set. The window constraint routines (see Table 2-1) affect only the *interactive* reshaping or moving of windows. When you use `winmove` to move a window, it does not have to stay in the same position forever. Later, if you interactively move it (i.e., by using pop-up menus), it is subject to the constraints that were in effect at the last call to `winconstraints`.

winpop

`winpop` pops the current graphics window in front of all other windows and icons on the screen. When you pop a window, it covers all the windows that occupy the same portion of the screen.

```
void winpop()

subroutine winpop
```

winposition

`winposition` sets the position and size (in pixels) of the current graphics window to the coordinates *(x1, y1)* and *(x2, y2)*. You can specify any two corners with the two *x* and two *y* coordinates. The 4Sight server automatically erases the window at the old position. `winposition` act on the current graphics window despite any constraints you have set. The window constraint routines (see
Table 2-1) affect only the *interactive* reshaping or moving of windows. When you use `winposition` to move a window, it does not have to stay in the same position forever.

```
void winposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

subroutine winpos(x1, x2, y1, y2)
integer*4 x1, x2, y1, y2
```

winpush

`winpush` pushes the current graphics window behind all other windows and icons on the screen. When you push a window, all overlapping windows obscure it.

```
void winpush()

subroutine winpus
```

2.5 Other Window Routines

This section describes the graphics window routines shown in Table 2-3.

Routine	Task
`endfullscrn`	End full-screen mode
`fullscrn`	Enable the entire screen for writing
`getorigin`	Return the origin of the graphics window
`getsize`	Return the size of a graphics window
`icontitle`	Give a window's icon a title
`reshapeviewport`	Put viewport at current graphics window position
`screenspace`	Put the program in screen space
`windepth`	Return depth of window in window stack
`winget`	Return identifier of current graphics window
`winset`	Set the current graphics window
`wintitle`	Make a title bar for the current graphics window

Table 2-3. Miscellaneous Window Routines

endfullscrn

`endfullscrn` ends full screen mode. The screenmask and viewport are reset to the boundaries of the current graphics window. The current transformation is unchanged. See the *Graphics Library Programming Guide*, Chapter 7, ‘‘Coordinate Transformations’’.

```
void endfullscrn()

subroutine endful
```

fullscrn

`fullscrn` enables the entire screen for writing. It sets the screenmask and the viewport to the entire screen, with respect to the physical screen origin, not the origin of the graphics window. When you use this routine, graphics processes are not protected against interference from each other.

```
void fullscrn()

subroutine fullsc
```

getorigin

`getorigin` returns the position of the origin (lower-left corner) of the graphics window.

```
void getorigin(x, y)
long *x, *y;

subroutine getori(x, y)
integer*4 x, y
```

getsize

`getsize` returns the size of the graphics window in pixels.

```
void getsize(x, y)
long *x, *y;

subroutine getsiz(x, y)
integer*4 x, y
```

icontitle

`icontitle` sets the title of the current window's icon. The title is center-justified, and appears near the top edge of the icon. The extra FORTRAN argument, *length* is the number of characters in *name.*

```
void icontitle(name)
char name[];

subroutine iconti(name, length)
character *(*)
integer*4 length
```

reshapeviewport

`reshapeviewport` sets the viewport to the current dimensions of the graphics window.

```
void reshapeviewport()
```

```
subroutine reshap
```

screenspace

`screenspace` puts the window in screen space. Graphics positions are expressed in absolute screen coordinates. (Normally, when you open a window, (0,0) is the lower-left corner of the window, rather than the lower-left corner of the screen.) In screen space, (0,0) is the lower-left corner of the screen. To get the upper-right corner of the screen, call the GL routine `getgdesc`. Call `getgdesc(GD_XPMAX)` for the the x coordinate. Call `getgdesc(GD_YPMAX)` for the y coordinate. Putting the program in screen space allows you to read pixels and locations outside the graphics window. (Note that when doing this, the user must maintain input focus to the window as described in section 2.1.)

```
void screenspace()
```

```
subroutine screen
```

windepth

`windepth` measures how deep a window is in the window stack. To determine the relative stacking order of windows on the screen, compare the returned value of this function for the various windows. The larger the value returned by `windepth`, the deeper the window is on the stack. *gid* is the graphics id number of the window whose depth you want to measure.

```
long windepth(gid)
long gid;
```

```
integer*4 function windep(gid)
integer*4 gid
```

winget

`winget` returns the graphics window identifier (*gid*) of the current graphics window.

```
long winget()

integer*4 function winget()
```

winset

`winset` makes the specified graphics window the current window. *gid* is the window's graphics id number. When a new current window is set, most graphics modes (such as `color`, `setpattern`, and `zbuffer`) are changed to reflect the graphics environment of the new current window.

For information on using `winset` with the Distributed Graphics Library (DGL), see Chapter 6, ''Using the Distributed Graphics Library''.

```
void winset(gid)
long gid;

subroutine winset(gid)
integer*4 gid
```

wintitle

`wintitle` makes a title bar for the current graphics window, which you can set with `winopen`.

Calling `wintitle("")` removes the title bar of the current graphics window. The *name* argument is displayed on the left-hand side of the title bar. The extra FORTRAN argument, *length*, is the number of characters in *name*.

```
void wintitle(name)
char name[];

subroutine wintit(name,length)
character*(*) name
integer*4 length
```

2.6 Subwindows

The GL Interface supports GL subwindows, borderless windows that are positioned and clipped relative to a parent GL window. Possible uses for GL subwindows include dialogue and alert boxes.

swinopen

`swinopen` creates a GL subwindow. *parent_id* is the *gid* of the parent subwindow. If an existing subwindow's *gid* is specified, the parent of that subwindow becomes the parent of the new subwindow for purposes of positioning (see below). The new subwindow inherits the current graphics window state, and becomes the new current window.

After calling `swinopen`, the application must call `winposition` to specify the location of the subwindow with respect to the origin of the parent window. Imaging in the subwindow is limited (clipped) to the area of the parent window.

When using the DGL (see Chapter 6, ''Using the Distributed Graphics Library''), the *parent_id* also identifies the graphics server associated with the window. The DGL directs all subsequent GL input and output to the server associated with *parent_id.*

`swinopen` queues INPUTCHANGE and REDRAW.

```
long swinopen(parent_id)
long parent_id;

integer*4 function swinop(parent)
integer*4 parent
```

2.7 Programming Hints

When you program under the window manager, be aware of the special considerations listed in the sections below.

2.7.1 Graphics Initialization

`winopen` initializes window graphics. Older IRIS products used a routine called `ginit`. `ginit` disables all window manager functions; only one such program may run at a time. Starting a new `ginit` program while another is running kills the first one. To maintain compatibility with older IRIS products, programs using `ginit` still work. However, you should now use `winopen` instead.

2.7.2 Shared Facilities

Although each graphics program is independent of the others, they must share the graphics resources. All IRIS-4D's support the simultaneous display of windows that use different display configurations (e.g., RGB with single-buffer, color index with double buffering, RGB with double buffering and overlay planes, etc.).

But the hardware imposes a limit on how many different types of windows you can open simultaneously. The hardware must maintain descriptions for each display configuration you are using. Fortunately, windows can share the display configurations, so you do not need a separate display configuration for each window.

The IRIS-4D (B and G models) can correctly handle the simultaneous display of up to three display configurations. The 8-bitplane Personal Iris can handle up to four, and the full-bitplane Personal Iris and the IRIS-4D GT can simultaneously display up to 16 different display configurations.

If you try to use more types of window than you have display configurations, the system does the best is can by taking resources from the display configuration for the least recently touched window. As a result, the display for some "old" window might change color or look a little strange. However, if you access that window again, it takes back the resources it needs)emat the expense of some other window display configuration.

In addition to sharing window display configurations, the system also requires that graphics programs cooperate in their color map usage if the programs run simultaneously and use color index mode. Likewise, windows created by a single program must share line patterns, raster fonts, and cursor glyphs.

2.7.3 The Event Queue

Each graphics program has an event queue (see the *Graphics Library Programming Guide*, Section 5.3, "The Event Queue." A REDRAW event appears in the queue under these circumstances:

- The user moves or reshapes a window.
- Part of a window is uncovered.
- The display mode changes.

An INPUTCHANGE event appears on the queue when a user directs the input focus to a new window or to the background. The value associated with the INPUTCHANGE event is the *gid* (graphics window identifier) of the window that has input focus, or 0 if the input focus is removed.

2.7.4 Window Manager Devices

The devices listed below are associated with window manager events. See the *Graphics Library Programming Guide*, Section 5.3, "The Event Queue", for more information on how to use devices.

REDRAW	the window manager inserts a redraw event each time a window needs to be redrawn. The REDRAW device is queued automatically.
REDRAWICONIC	queues automatically when `iconsize` is called. The Window Manager sends this event when a window needs to be redrawn *as an icon* by the program itself.
DEPTHCHANGE	indicates an open window has been pushed or popped. The value of the event is the gid of the window that has changed. Use `windepth` to determine the stacking order.

WINSHUT	when queued, the Window Manager sends this event when the Close item is selected from a program's Window (frame) menu, or when the close fixture is selected from the title bar of a program's window. If WINSHUT is not queued, the Close item on the program's Window menu will appear greyed-out, and will have no effect if selected.
WINQUIT	when queued, the Window Manager sends this event rather than killing a process when Quit is selected from a program's Window (frame) menu.
WINFREEZE/WINTHAW	if queued, the Window Manager sends these events when windows are stowed to icons and later unstowed, rather than blocking the processes of the stowed windows. These devices should be queued if the program plans to draw its own icon (see `iconsize`) or is a multi-window application.
INPUTCHANGE	indicates a change in the input focus. The INPUTCHANGE device is queued automatically.

2.8 Sample Program: Single Buffer Mode

This section contains a program that runs under the window manager in single buffer mode. `keepaspect`, `winopen`, and `reshapeviewport` are the only window manager routines required.

The *drawit* routine executes once. Then, whenever the user interactively moves the window, *drawit* executes again. (Because of `keepaspect`, the window cannot be reshaped.) `reshapeviewport` makes sure the viewport is set to the new location of the current graphics window.

■ C Program: SINGLE BUFFER MODE

```
#include <gl/gl.h>
#include <gl/device.h>

main()
{
    short val;

    keepaspect(1,1);  /* the graphics window can be any
                location and size, as long as it's square */
    winopen("zoing");
    drawit();      /* image drawn the first time */
        /* the image is redrawn whenever a REDRAW
            appears in the event queue */
    while(TRUE) {
         if(qread(&val) == REDRAW) drawit();
    }
  /* NOT REACHED */
}

drawit()
{
    register int i;

    reshapeviewport();
    color(WHITE);
    clear();
    ortho2(-1.0,1.0,-1.0,1.0);
    color(BLACK);
    translate(-0.1,0.0,0.0);
    pushmatrix();
    for(i=0; i<200; i++)  {
        rotate(170,'z');
        scale(0.96,0.96,0.0);
        pushmatrix();
        translate(0.10,0.0,0.0);
        circ(0.0,0.0,1.0);
        popmatrix();
    }
    popmatrix();
}
```

2.9 Sample Program: Double Buffer Mode

This double buffered program draws a cube that the user rotates by moving the mouse.

■ C Program: DOUBLE BUFFER MODE

```
#include <gl/gl.h>
#include <gl/device.h>

main()
{
    Angle x, y;         /*  current rotation of object*/
    short active;       /*  TRUE if window is attached*/
    Device dev;
    short val;

    keepaspect(3,2);
    winopen("cube");
    doublebuffer();
    gconfig();
    qdevice(WINSHUT);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
    perspective(400, 3.0/2.0, 0.001, 100000.0);
    translate(0.0, 0.0, -3.0);
    rotate(900, 'z');

    x = 0;  y = 0;
    active = 0;
    while(TRUE) {
        drawcube(x,y);     /*  draw into the back buffer */
        swapbuffers();     /*  show it in the front buffer */
        while (qtest()) { /*  process queued events */
            dev = qread(&val);
            switch(dev) {
                case WINSHUT: /*  exit program */
                    gexit();
                    exit(0);
                    break;
                case INPUTCHANGE:
                    active = val;
                    break;
                case REDRAW:
                    reshapeviewport();
                    break;
                case MOUSEX:
                    x = val;
                    break;
                case MOUSEY:
                    y = val;
                    break;
                default:
                    break;
            }
        }
    }
  /* NOT REACHED */
}
```

```
drawcube(rotx,roty)
Angle rotx, roty;
{
    color(BLACK);
    clear();
    color(WHITE);
    pushmatrix();
    rotate(rotx,'x');
    rotate(roty,'y');
    cube();
    scale(0.3,0.3,0.3);
    cube();
    popmatrix();
}

cube() /* make a cube out of 4 squares */
{
    pushmatrix();
    side();
    rotate(900,'x');
    side();
    rotate(900,'x');
    side();
    rotate(900,'x');
    side();
    popmatrix();
}

side() /* make a square translated 0.5 in the z direction */
{
    pushmatrix();
    translate(0.0 0.0,0.5);
    rect(-0.5,-0.5 0.5,0.5);
    popmatrix();
}
```

3. Making Pop-up Menus

This chapter describes routines that your graphics programs can use to create and interact with pop-up menus. When the user selects an item from a menu, these routines automatically identify which menu item has been selected.

Sections 3.1 and 3.2 describe the routines that create and call pop-up menus. Section 3.3 describes advanced menu formats. Sections 3.4 and 3.5 provide examples programs that use pop-up menus.

Table 3-1 correlates tasks with pop-up menu routines.

Routine	Task
`addtopup`	Add menu items
`defpup`	Make a menu with items
`dopup`	Call up a menu
`freepup`	Delete a menu
`newpup`	Make a new menu
`setpup`	Enable or disable menu entries

Table 3-1. Pop-up Menu Routines

3.1 Defining a Pop-up Menu

If you program in C, you can define a menu in two ways:

- Initialize the menu definition with `newpup`, and then add menu items to the definition by calling `addtopup`.
- Use `defpup`, which combines the functions of `newpup` and `addtopup`.

If you program in FORTRAN, you must use `newpup` and `addtop` to define your menu. `defpup` is available only to C programs.

newpup

`newpup` allocates and initializes a structure for a new menu. This function takes no arguments and returns a 32-bit integer identifier (*pup*) for the pop-up menu.

```
long newpup()

integer*4 function newpup()
```

addtopup

`newpup` defines an empty menu. To build a menu definition, use both `newpup` and `addtopup`. `addtopup` adds menu entries to the bottom of an existing menu definition (i.e., the menu definition created by `newpup`).

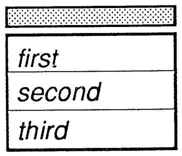
pup is the menu identifier returned by `newpup` or `defpup`. *str* is a character string that specifies the entries in the menu. The string lists the menu labels from the top to the bottom of the menu. Use a '|' (vertical bar) between entries.

The FORTRAN version, `addtop`, takes an additional *length* argument, which specifies the number of characters in the string. Use *arg* for advanced menu formats (see Section 3.3). If you are working with a simple menu, you can supply a zero for *arg*.

```
void addtopup(pup, str, arg)
long pup;
char *str;
long arg;

subroutine addtop(pup, str, length, arg)
integer*4 pup
character* str(*)
integer*4 length
integer*4 arg
```

To define a menu that looks something like this:



use these routines:

C:

```
menu = newpup();
addtopup(menu, "first|second|third", 0);
```

FORTRAN:

```
imenu = newpup()
call addtop(imenu, "first|second|third", 18, 0)
```

The *18* in the FORTRAN routine is the number of characters in the string, including the vertical bars.

defpup

`defpup` is available in C only. Use `defpup` to define a new menu its entries. `defpup` combines the functions of `newpup` and `addtopup`.

```
long defpup(str [, args] ...)
char *str;
long args;
```

`defpup` creates the menu shown above in one step. You can add additional menu entries with `addtopup`.

```
menu = defpup("first|second|third");
```

setpup

`setpup` allows you to disable individual entries in a menu. The disabled entry appears greyed out when the system displays the menu. A greyed out entry returns no value if the user selects it. *pup* is the menu identifier returned by either `newpup` or `defpup`. *entry* is the position of the menu entry, indexed from 1. *mode* determines the display state of the chosen entry. See the `setpup` man page for more information on menu items display modes.

```
void setpup(pup, entry, mode)
long pup;
long entry;
long mode;

subroutine setpu(pup, entry, mode)
integer pup
integer entry
integer mode
```

3.2 Calling up a Pop-up Menu

This section describes how a GL client can display previously-defined pop-up menus. This section also describes how to free the memory associated with a menu definition.

dopup

`dopup` displays a previously defined pop-up menu. *pup* is the identifier of the pop-up menu. The code below uses `qread` and `dopup` to display a menu while the user presses the right mouse button. If the user releases the button while the cursor is off the menu, `dopup` returns –1. If the user releases the button while the cursor is on the menu and if the menu definition does not use an advanced menu format (see Section 3.3), `dopup` returns an integer corresponding to the position of the item in the menu.

```
long dopup(pup)
long pup;

subroutine dopup(pup)
integer*4 pup
```

To cause the right mouse button to bring up the menu shown in Section 3.1, use this code:

C:

```
dev = qread(&val);
if (dev == RIGHTMOUSE) {
    if (val == 1) {       /*  right mouse button is pressed  */
        menuval = dopup(menu);
    }
}
```

FORTRAN:

```
idev = qread(ival)
if (idev .eq. rightm) then
    if (ival .eq. 1) then
        imval = dopup(imenu);
    endif
endif
```

The user selects ‘first’, ‘second’, or ‘third’ by positioning the cursor over one of these items, then releasing the button. The user makes no selection by releasing the button with the cursor off the menu. Table 3-2 shows the return value for each possible selection.

Selection	Return Value
first	1
second	2
third	3
no selection	–1

Table 3-2. Default Return Values

To make `dopup` return values other than the position value of a selected menu item, see Section 3.3.

freepup

`freepup` frees the memory used to define a pop-up menu. Freeing this memory deletes the pop-up menu definition.

```
void freepup(pup)
long pup;

subroutine freepu(pup)
integer*4 pup
```

3.3 Advanced Menu Formats

Until now, the strings used in `addtopup` or `defpup` to define menu items were simple. The only special formatting character mentioned has been the "|" delimiter. This character separates the defining text for multiple menu items (e.g., "first|second|third"). However, there are special character combinations that allow you to:

- define the value returned when you select a menu item
- bind a function to a whole menu or to a menu item
- make a title bar or a rollover menu

You can use these special character combinations with either the `addtopup` or the `defpup` routine. The general rules for using these special character combinations are:

- Put the special character combination at the end of the menu item to which it applies (before a "|" delimiter, or, if the string defines only one menu item, before the closing quote).
- Start all special character combinations with a percent sign "%".
- Put only one command letter after the percent sign. If you want to use more than one command letter in a single menu item, use a % before each (e.g. %t %f).
- Put numeric arguments (if any) immediately after the command letter (e.g., %x15). If the command takes an argument that is not a number (%f takes a routine name), use the *arg* parameter of the `addtopup` or `defpup` routine to specify the non-numeric argument.

You can use more than one special character combination in a single menu entry. Because there is only one *arg* parameter in the `addtopup` or `defpup` routine, *string* can contain only one command that takes a non-numeric argument. To get around this limitation, you can make multiple calls to `addtopup` when you define the menu. Below, Table 3-3 summarizes the special character combinations. (Detailed descriptions of each special character combination follow the table.)

Command	**Task**	**Changes return value?**	**Takes numeric value?**	**Takes the *arg* Parameter?**
%f	Bind function/menu item	yes	optional	yes
%F	Bind function/whole menu	yes	optional	yes
%l	Underline menu item	no	no	no
%m	Make nested menu	yes	no	yes
%n	Return default values	no	no	no
%t	Make title	no	no	no
%x	Return other values	yes	yes	no

Table 3-3. Summary of Advanced Menu Formats

3.3.1 Binding a Function to a Menu Entry, f

%f is similar to %F, but %f affects the value returned for its particular menu item. %f requires an argument: the function that generates the value that `dopup` returns for this menu item. You name this function as the *arg* parameter of `addtopup` or `defpup`.

```
menu = newpup();
addtopup(menu, "first|call %f", funct);
```

Selecting "first" causes `dopup` to return the value of 1. Selecting "call" causes `dopup` to return the value of `funct(2)`. Call `addtopup` each time you want to add another menu entry that has its own function.

3.3.2 Binding a Function to a Whole Menu, F

Use %F to specify a function that affects all values returned by all items in the menu. %F requires an argument: the function that generates the value that `dopup` returns. You name this function as the *arg* parameter of `addtopup` or `defpup`.

```
menu = newpup();
addtopup(menu, "Cardinal %t %F|first|second %x10", funct);
```

Selecting "first" causes `dopup` to return the value of `funct(1)`, instead of 1. Selecting "second" causes `dopup` to return the value of `funct(10)`.

3.3.3 Drawing a Line Under a Menu Item, l

Use %l to draw a line under a menu item. You can use this line to mark off related entries in a menu. For example:

```
menu = newpup();
addtopup(menu, "1st|2nd|3rd %l|None", 0);
```

This code creates a menu with a line under the item "3rd".

3.3.4 Making a Nested (Rollover) Menu, m

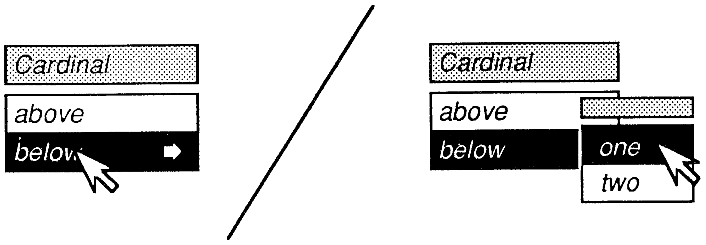
Use %m to create a simple nested pop-up menu (a submenu). When you roll the cursor to the right side of the menu item, you invoke the submenu. The submenu is like any other pop-up menu and can contain a number of selections. %m requires an argument: the menu identifier for the pop-up submenu. Supply the menu identifier as the value of the *arg* parameter of either `addtopup` or `defpup`.

Note: Because %m takes the submenu's menu identifier as an argument, your code must define the submenu before it defines the main menu. Defining a submenu is the same as defining any other menu. Call `newpup` to get the menu identifier, then call `addtopup` to add items to the menu.

The code:

```
submenu = newpup();
addtopup(submenu, "one|two", 0);
menu = newpup();
addtopup(menu, "Cardinal %t|above %x5|below %m", submenu);
```

creates these menus:



If you select an item from the submenu, `dopup(menu)` returns the same value as `dopup(submenu)` would. If you display the submenu but don't select anything from it, or simply select the menu item without rolling over, dopup returns –1.

3.3.5 Resetting to Default Values, n

Use %n to return a menu entry to its default settings. %n takes no arguments. The menu:

```
menu = newpup();
addtopup(menu, "first|second|third", 0);
```

is the same as the menu:

```
menu = newpup();
addtopup(menu, "first %n|second %n|third %n", 0);
```

3.3.6 Making a Title Bar, t

Use %t to create a title bar on a pop-up menu. You cannot select the title bar; it does not highlight.

```
menu = newpup();
addtopup(menu, "Cardinal %t|first|second|third", 0);
```

This is the same as the first example above, except there is a title bar at the top of the menu. %t takes no arguments.

3.3.7 Setting the Value of a Selection, x

Use %x to change the numeric value that `dopup` returns when the user selects a menu item. For the menu:

```
menu = newpup();
addtopup(menu, "first %x15|second %x7", 0);
```

Selecting 'first' causes `dopup` to return 15, not 1. Selecting 'second' causes `dopup` to return 7, not 2. When you use %x, you must specify the numeric value that `dopup` returns in place of the default.

3.4 An Example from *cedit*, a Color Editing Program

cedit is a simple color editor that runs under the window manager. It is available on your IRIS-4D Series workstation, in the directory */usr/sbin.*

The program brings up a window containing a color control system in which red, green, and blue sliding bars change the color components of a sample patch of color. The right mouse button brings up a menu that allows you to choose from four color systems; the left mouse button selects a color on the screen to edit and also controls the sliding bars.

The *cedit* window and menu are created by three lines of code:

```
keepaspect(1,1);
winopen("cedit");
menu = defpup("colorsys %t|rgb|cmy|hsv|hls");
```

`keepaspect`, one of the window constraint routines discussed in Section 2.3, requests a square aspect ratio. You must call `keepaspect` before `winopen` so that the window manager enforces this constraint when it creates the window. The pop-up menu definition creates a menu with title 'colorsys' and menu entries 'rgb', 'cmy', 'hsv', and 'hls'.

The code sample below sets the color system according to the your menu selection.

```
dev = qread(&val);
switch (dev) {
case MENUBUTTON:
    sel = dopup(menu);
    if (sel > 0) {
        setcolorsys(sel);
        newcolor(cc);
    }
       break;
}
```

When you make a selection from the pop-up menu by using the right mouse button, `dopup(menu)` returns a value. Because the pop-up menu definition does not include any advanced menu formats, the return value is the default: 1 for the first selection ('rgb'), 2 for the second ('cmy'), and so on. The return value is –1 if you don't make any selection at all.

The program then assigns the return value to the variable *sel.* If *sel* is greater than zero (i.e., if you make a selection), then the program sets the color system accordingly, and calls *newcolor*, a routine defined in the *cedit* program. *newcolor* sets the color of the sample patch, and sets the sliding bars to the correct position for that color within the selected color system.

The case statement started in the code sample above continues with instructions for the middle and left mouse buttons. If you point to a color outside the *cedit* window, a click of the middle mouse button changes that color to the color of the sample patch. A click of the left mouse button retrieves one pixel and makes the color of that pixel the current color of the color patch, and you can edit it.

If you know which color system you want to use for editing a color, you can bypass the 'colorsys' menu by giving *cedit* an argument. The argument corresponds to the position of the color system in the 'colorsys' menu. The *cedit* window then appears with the color system already set.

The code fragment shown below, which makes this possible, immediately precedes the window constraint and `winopen` routines.

```
main(argc,argv)
int argc;
char *argv[];
{
    int i, j;

    if (argc > 1)
        setcolorsys(atoi(argv[1]));
```

3.5 Sample Program

This program demonstrates the use of the pop-up menu routines described in this chapter. The left mouse button puts shapes on the screen if you direct input focus to the window entitled *blop*. Choose 'Quit' from the Window menu to terminate the program.

■ C Program: POP-UP MENU

```
#include <gl/gl.h>
#include <gl/device.h>

#define CIRCLE 1
#define RECT   2
#define LINE   3

int mainmenu;
int colormenu;
int shapemenu;
int curcolor;
int curshape;
int xorg, yorg;

setshape(n)
int n;
{
    curshape = n;
    return -1;     /* dopup returns this value */
}

setcolor(n)
int n;
{
    curcolor = n;
    return 7;      /* dopup returns this value */
}
```

```
main()
{
    short val;
    int x, y;
    int pupval;

    prefsize(400,300);
    winopen("blop");
    curcolor = WHITE;
    curshape = CIRCLE;
    qdevice(LEFTMOUSE);
    qdevice(MENUBUTTON);
    shapemenu = defpup("shapes %t %F|Circle|Rect|Line",setshape);
    colormenu = defpup("colors %t %F|BLUE %x4|WHITE %x7|RED %x1",
                       setcolor);

    mainmenu = defpup("blop %t|shapes %m|color %m|clear|set",
                      shapemenu,colormenu);
    makeframe();
    while(TRUE) {
      switch(qread(&val)) {
          case REDRAW:
              makeframe();
              break;
          case LEFTMOUSE:
              if(val) {
                  x = getvaluator(MOUSEX)-xorg;
                  y = getvaluator(MOUSEY)-yorg;
                  drawshape(curcolor,curshape,x,y);
              }
              break;
          case MENUBUTTON:
              if(val) {
                  pupval = dopup(mainmenu);
                  switch(pupval) {
                    case 3:                   /* clear */
                        color(BLACK);
                        clear();
                        break;
                    case 4:             /* set */
                        color(WHITE);
                        clear();
                        break;
                  }
              }
        }
    }
}

drawshape(acolor,ashape,x,y)
int acolor, ashape, x, y;
{
    color(acolor);
    switch(ashape) {
    case CIRCLE:
        circfi(x,y,10);
        break;
    case RECT:
        rectfi(x,y,x+15,y+10);
        break;
    case LINE:
        move2i(x,y);
        draw2i(x+20,y+20);
        break;
    }
}

makeframe()
{
    reshapeviewport();
    getorigin(&xorg,&yorg);
    color(BLACK);
    clear();
}
```

4. Controlling Multiple Windows from a Single Process

This chapter describes methods that allow you to create and control multiple windows from a single process. A typical example might be a two-window program consisting of a static control panel window and a dynamic display window. A different type of example would be a multi-windowed editor, that lets you edit many documents, each in a separate window.

4.1 Using Window Identifiers

The window manager associates a unique integer name, called the *gid* (graphics window identifier), with each window as it is created. This number allows you to keep track of the window. Window identifiers are not guaranteed to be unique among different processes, but are always unique within a given process.

The 4Sight window manager keeps track of the windows that a process opens. It responds to different UI events by placing various events on the GL event queue associated with the process that owns the windows. For example, if the user uncovers (pops) an open window, the window manager sends a REDRAW event and the window's *gid* to the process that created the window. Your program should use the information it receives from the event queue to redisplay windows that have been changed by the user. When your program responds to the REDRAW event, it should always call `reshapeviewport` in case the window has been resized.

The code segments below show how to control multiple windows. The entire program is reprinted at the end of this chapter.

To create the windows, call `winopen`:

```
for (i = 1; i <= nwins; i++) {
        ...
        wins[i].gid = winopen ("MultiWin");
        ...
}
```

The *wins* array of structures saves the *gid* for each window. The current graphics window is the window in which drawing and window manipulation take place. Only one graphics window is current for any process. `winopen` makes a newly opened window the current graphics window.

To draw into a particular window, use the `winset` routine to select a window as the current window. The program fragment below sets the current window to *val* and then clears that window to its current color.

```
winset(val); /* set to window that needs redrawing */
color(wins[whichwindow(val)].color);
clear();
```

Each window has its own graphics state. This state includes the matrix stack, single versus double buffered mode, RGB versus color index mode, and all the pushable attributes. Exception for `winset`, if a routine effects the graphics state, it does so for the current graphics window only. Routines like `qdevice` don't effect the graphics state.

4.2 Example Program

The following program calls `qdevice` for all pseudo devices that are important from the standpoint of a multi-windowed application. The comments in the code describe what a program should do when it receives these types of events.

■ C Program: MultiWin

```
/*******************************************************************
    MultiWin --- program of multiple windows from a single process.

    To compile type:
    cc -o multiwin multiwin.c -lgl_s
    To run type:
    multiwin 3
    The argument determines how many windows to open

    The left mouse button cycles the color used to redraw the
    background of each window.
*******************************************************************/
#include "gl.h"
#include "device.h"

/* scale factor associated with each window id */
static float scalewin[50];

main (argc, argv)
int     argc;
char  *argv[];
{
    Icoord   x;
    short    i, oldx, active;
    register dx;
    short    val;
    Angle    cumex, cumey, cumez; /*  rotation angles   */
    int     nwins;    /*  how many windows for the cube    */
    int     win_ids[10]; /*  window ids 10 windows per process.  */
    int     multiwindow;

    if (argc < 2) {
        printf ("Usage: multicube numberofwindows0);
        exit (0);
    }
    if (!ismex()) {
        printf ("You must be in the window manager to run multicube0);
        exit(0);
    }
    cumex = 0; cumey = 0; cumez = 0;
    active = FALSE;

    sscanf (argv[1], "%d", &nwins); /*  count how many windows to open*/

/*  limit number of windows to somewhere between 1 and 10  */
    if (nwins <= 0) {
        printf ("It is meaningless to open fewer than one window\n");
        printf ("This program assumes that you want to open 1 window\n");
        nwins = 1;
    }
    else if (nwins > 10) {
        printf ("At most, you can open ten windows per process\n");
        nwins = 10;
    }

    foreground();
/*  open all the windows */
    for (i = 0; i < nwins; i++) {
        win_ids[i] = winopen ("cubes");
    doublebuffer();
    gconfig();
    perspective (900, 1.0, 1.0, 3.5);
    translate (0.0, 0.0, -2.0);
    }
    qdevice (REDRAW);
    qdevice (INPUTCHANGE);
    qdevice (LEFTMOUSE);
    qdevice (MIDDLEMOUSE);
    qdevice (MOUSEX);
    qdevice (ESCKEY);


/*  for each window, calculate the scale factor, * /
/*  and then draw initial scene */

    for (i = 0; i < nwins; i++) {
        scalewin[win_ids[i]] = 1.0/(float) (i + 1);
        qenter(REDRAW,win_ids[i]);
    }
```

```
while (1) {

    if (qtest ()) {
        switch (qread (&val)) {
            case ESCKEY:
                gexit();
                exit(0);
                break;
            case REDRAW:
  /* redraw only those windows that need to be redrawn*/
                winset ((int) val);
                reshapeviewport ();
            pushmatrix();
            cube (WHITE, cumex, cumey, cumez, val);
            popmatrix();
            break;
            case INPUTCHANGE:
            if (val)
            active = TRUE;
            else {
                active = FALSE;
                for (i = 0; i < nwins; i++) {
                    winset (win_ids[i]);
                    pushmatrix ();
                    cube (WHITE, cumex, cumey, cumez, win_ids[i]);
                    popmatrix ();
                }
            }
                break;
         case MOUSEX:
             oldx = x;
             x = val;
             dx = x - oldx;
             if (getbutton(LEFTMOUSE))          /* rotate x */
                 cumex = cumex + dx;
             else if (getbutton(MIDDLEMOUSE))   /* rotate y */
                 cumey = cumey + dx;
             else if (getbutton(RIGHTMOUSE))    /* rotate z */
                 cumez = cumez + dx;
             break;
            default:
                break;
                }
            }
        for (i = 0; i < nwins; i++) {
            winset (win_ids[i]);
            pushmatrix ();
            cube (WHITE, cumex, cumey, cumez, win_ids[i]);
            popmatrix ();
        swapbuffers();
    }                               /* end if(qtest()) */
    }                                       /* end while */
}

/*  draw cube with color, rotation and scale factor        */

cube (hue, cumex, cumey, cumez, winid)
Colorindex hue;
Angle cumex, cumey, cumez;
int winid;
{
    color (BLACK);
    clear ();
    rotate (cumex, 'x');
    rotate (cumey, 'y');
    rotate (cumez, 'z');
    scale (scalewin[winid], scalewin[winid], scalewin[winid]);
    color(hue);
    move (-1.0, -1.0, -1.0);
    draw (-1.0, -1.0, 1.0);
    draw (1.0, -1.0, 1.0);
    draw (1.0, 1.0, 1.0);
    draw (-1.0, 1.0, 1.0);
    draw (-1.0, -1.0, 1.0);
    move (-1.0, -1.0, -1.0);
    draw (-1.0, 1.0, -1.0);
    draw (1.0, 1.0, -1.0);
    draw (1.0, -1.0, -1.0);
    draw (-1.0, -1.0, -1.0);
    move (1.0, -1.0, -1.0);
    draw (1.0, -1.0, 1.0);
    move (-1.0, 1.0, -1.0);
    draw (-1.0, 1.0, 1.0);
    move (1.0, 1.0, -1.0);
    draw (1.0, 1.0, 1.0);
    move (1.0, 1.0, 0.0);
    draw (-1.0, 1.0, 0.0);
    draw (-1.0, -1.0, 0.0);
    draw (1.0, -1.0, 0.0);
    draw (1.0, 1.0, 0.0);
}
```

4.3 Managing Multiple Monitors and Screens

A monitor is a tangible device, a tube. For the purposes of this discussion, a screen is a contiguously addressed two-dimensional pixel space. In general, there are three ways according to which computer systems can provide for multiple monitors:

- one frame buffer displayed on all monitors (one screen repeated on each monitor)
- separate frame buffers for each screen with discontiguous addressing across them (one screen per monitor)
- separate frame buffers for each monitor with contiguous addressing across them all (one screen spread across several monitors— not supported on Silicon Graphics systems)

Note: Because the addressing to the frame buffers is not contiguous on Silicon Graphics systems, it is not possible for windows to span from one screen to the other.

Screens are numbered from 0 to the returned value of `getgdesc(GD_NSCRNS)` minus 1.

One Screen Repeated on Multiple Monitors

If the monitors are of a different resolution or video timings, the image does not look right on both monitors simultaneously. This is acceptable when proofing output to video tape (NTSC) because you can perfect the image on the high resolution monitor and ignore the image on the low resolution monitor. You can then make a call to `setmonitor` to change to NTSC mode and view the image on the NTSC monitor before writing it to tape.

Separate Screens for Each Monitor

Silicon Graphics currently offers the M4D/20 with an auxiliary monitor. The auxiliary monitor has a frame buffer that is independent of the frame buffer for the main monitor. The images displayed on each monitor are independent of each other, and the frame addresses are not contiguous so a window cannot span monitors.

The input devices (the event queue) are system-wide and therefore your program receives input from all the windows it opened—whether it opened those windows on the auxiliary or master monitor. However, the windowing environment on the auxiliary monitor differs considerablely from that on the master monitor.

4.3.1 The Behavior of Windows on Auxiliary Screen

In the current release, window management is provided only on the master monitor, screen 0. There is no window management on the auxiliary screens. The implications of no window management on auxiliary screens are:

- There can be only one window on an auxiliary screen. `winopen` fails (returns –1) if you try to open a second.
- Once opened, a window can not be moved to a different screen.
- `windepth` always returns *top depth* for an auxiliary screen window.
- The appearance of pop-up menus on an auxiliary screen differs from their appearance on the main screen.
- All Window Manager hints (e.g. `prefsize`) are ignored for the auxiliary screen window.
- A window on an auxiliary screen ignores calls to `swinopen`, `winconstraints`, `winmove`, `winpop`, `winposition`, `winpush`, `wintitle`, `iconsize`, and `icontitle`.

4.3.2 Switching Input Focus to Another Screen

There is no window manager feature that the user can use directly to switch input focus to another screen. To get around this, you need to create a button/icon or some other device that your program can monitor for input from the user. When the user signals for a screen change, call `scrnattach`.

A process can attach to a window on an auxiliary screen only if it was the process that created the window. That is, you can call `scrnattach(1)` only if yours was the process that did the:

```
scrnselect(1);
winopen( ... );
```

To get the number of the currently selected screen, call the function `getwscrn`. The returned value of this function is the screen number for the currently selected screen.

5. Using the IRIS Font Manager

The IRIS Font Manager is a service (available to C programs only) that lets you render characters from specific fonts. The Font Manager also lets you query the system about those fonts. A *font* is a group of character representations (glyphs) of a given typeface in a particular weight and inclination. The naming conventions for Font Manager fonts parallel those used in the PostScript programming language and NeWS.

For example, a bold, italic face in Times Roman is called ''TimesRoman-BoldOblique''. The Font Manager adopts the NeWS method of offering a bitmap representation for a selected number of sizes of a font-face: selecting the closest one appropriate to a request and adjusting character set widths accordingly. Under special user-controllable circumstances, the Font Manager creates an exact size (see the discussion of `fmrotatepagematrix`, below).

All clients of the window system use the Font Manager, including NeWS and Graphics Library clients. The public interface to the Font Manager is defined in the file */usr/include/fmclient.h.* A GL program can access the Font Manager by linking with */usr/lib/libfm.a.* To make this link, use the **–lfm** switch on the loader line. For multi-process code, there is a shared library version of the Font Manager. To link a GL program with this version of the Font Manager, use the **–lfm_s** switch.

NeWS clients continue to use their respective systems as before: there are no new interfaces except new switches used in programs for converting fonts created by third-party suppliers (see section 5.1.3, ''Importing Fonts'').

If you have used the Graphics Library routine `defrasterfont` to create fonts, you can continue to do so; its data format and imaging do not involve the Font Manager. To maintain compatibility with `defrasterfont`, the current character position is maintained when the Font Manager renders strings. (See `cmov` in the *Graphics Library Reference Manual*).

5.1 The Font Manager Interface

The closest paradigm to the way the Font Manager specifies fonts and renders text is PostScript. While the Font Manager differs from PostScript in many respects, many of its notions serve as a guide.

A *font family* is the set of files that make up a font. The characters defined in these files can vary in size and rotation, but not in font name, weight, or inclination.

5.1.1 Available Fonts

There are two ways to find out what fonts are available through the Font Manager. One is to write a program that calls `fmenumerate`. This routine returns a list of font families available through the Font Manager. The other is to list the contents of */usr/lib/fmfonts*, looking for any file ending in *.ff*.

If the font name is long, these filenames are truncated versions of the name of the font. To find the actual name of the font, run the program *strings* on a *.ff* file. The first line is the name of the font. Actual font data files do reside in this directory, but interpreting their names to discover the exact sizes of fonts available can be misleading (see below).

5.1.2 Device Independence

The Font Manager automatically transforms font size requests, depending on the resolution of the screen and the ''ideal'', or originating, or ''design'' resolution of the font. Font size specifications are made in points. Therefore, asking for a 14-point font on a 96 dpi screen can result in looking up the font data for an 18-pixel font. This happens if the fonts were designed for a 72-dpi screen (where 14 points is 14 pixels high). On a 96-dpi screen, the same 14-point font is only 3/4 the size it would be on a 72-dpi screen.

Therefore, the closest font (in size) is selected by applying the appropriate transformation. As mentioned above, this can lead to confusion when trying to use a listing of f2/usr/lib/fmfonts to determine which exact font sizes exist. Seeing a file in */usr/lib/fmfonts* whose name indicates a 14-point font does not necessarily mean you actually get that file when rendering 14-point text.

5.1.3 Importing Fonts

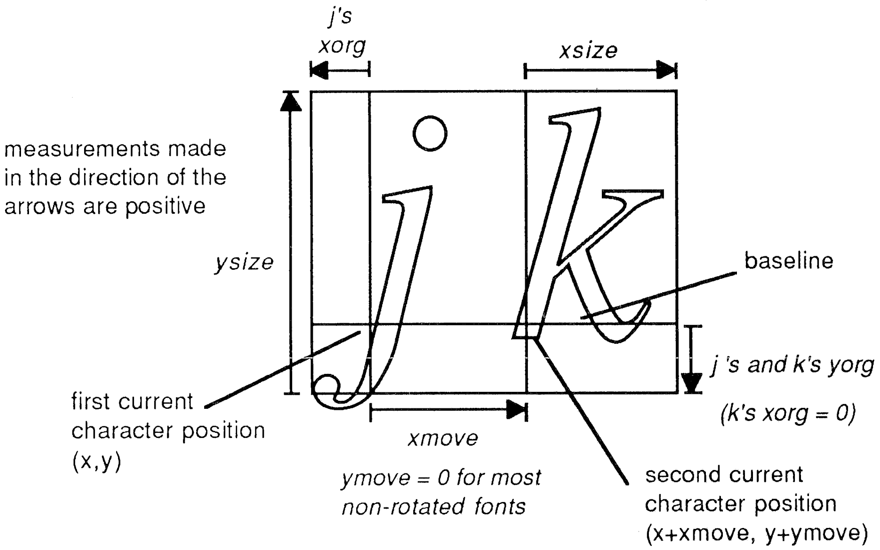
The Font Manager supports all the windowing systems and comes with a wide variety of fonts. You can add a screen font by purchasing it and using the *dumpfont* and *bldfamily* utilities to converting it to the Font Manager binary format. These conversion utilities can handle a few different input formats, but you should try to get the font in Adobe Binary Distribution Format. Refer to Appendix D of Section 2, ''Programming in NeWS'', as well as the *bldfamily*(1) and *dumpfont*(1) man pages.

5.1.4 Font Metrics

The metrics (dimensions) of a character are given in the **struct** *fmglyphinfo*:

```
typedef struct fmglyphinfo {
    long xsize, ysize;     /* dimensions of glyph in pixels */
    long xorig, yorig;     /* origin */
    float xmove, ymove;    /* move  */
    long gtype;            /* glyph type */
    long bitsdeep;         /* depth of pixels, if pixels */
} fmglyphinfo;
```

All but two character metrics are long integers. *xmove* and *ymove* are **floats**. The basic unit of the values in *xmove* and *ymove* is the device unit (pixel). By making *xmove* and *ymove* **floats**, the Font Manager supports the subpixel positioning information needed by typesetting and graphics applications. The value *yorig* is the distance from the bottom of the glyph to the baseline. The value *xorig* is the horizontal distance from the current character position to the left edge of the glyph; either can be a negative value. *xsize* and *ysize* are the character boundaries (for a bitmap glyph, this is the bitmap size).



j's
xorg
xsize
measurements made in the direction of the arrows are positive
ysize
baseline
j 's and k's yorg
(k's xorg = 0)
first current character position (x,y)
xmove
ymove = 0 for most non-rotated fonts
second current character position (x+xmove, y+ymove)

Figure 5-1. Font Manager Font Metrics

5.1.5 Font Specification and Sizing

As with PostScript, you must specify the font (by family and point size) before you can render a string of characters onto the screen. After calling `fminit`, call `fmfindfont("`*FamilyName*`")`. *This routine returns a handle to the specified font family. To get a specific font within that font family, call* `fmscalefont`. This routine expects a point size and the handle returned by `fmfindfont`. Using this information, scalefont returns a handle to a specific font. Using this handle, you can then call `fmprstr` to render characters in the specified font and point size. (More flexible ways of rendering text are discussed later.)

5.1.6 Font Transformation

The the following paragraphs describe the transformation paradigm used by the Font Manager. You do not need to explicitly modify the page matrix to print scaled text, but you do need to call `fmrotate` to print rotated text.

The Font Manager maintains an abstract notion of font rendering, called the *page*. Think of the page of as a transparent sheet that is superimposed on the current window. The page maintains a coordinate system for font rendering. Application programs can make calls to the Font Manager to modify the page's transformation matrix. Changing the page's transformation matrix changes the appearance of the font in the window. You can make calls to the Font Manager if you want to render scaled or rotated text. If you do not want to alter the page's transformation matrix, you can use `fmscalefont` to scale the characters and not the page..

Conceptually, there is a distinction between scaling a font and scaling the characters of a font as they are rendered. The following code first draws a one-point high string and then draws a string of two-point high characters. The text appears larger because the size of the page is doubled. The font is actually still a one-point high font, but the characters are scaled as they are rendered.

```
font1 = fmfindfont("Times-Roman");
fmsetfont(font1);
fmprstr("Hello");
fmscalepagematrix(2.0);
fmprstr("World");
```

You can produce an identical effect using the code below. The font drawn by the code is a true two-point high font, but the page scale is still at a 1:1 ratio. Calling `fmprstr` draws a string of two-point-high characters.

```
font1 = fmfindfont("Times-Roman");
fmsetfont(font1);
fmprstr("Hello");
font2 = fmscalefont(font1,2.0);
fmsetfont(font2);
fmprstr("World");
```

This illustrates the fact that both the font and the page have a transformation matrix. Before rendering, the font's transformation matrix is concatenated with the page's transformation matrix and the resultant font size is rendered onto the page. The font's transformation matrix is stored with the font. The page's transformation matrix is stored in the client's process space. To set or read the page's transformation matrix, use the routines:

```
fmconcatpagematrix
fmgetpagematrix
fminitpagematrix
fmrotatepagematrix
fmscalepagematrix
fmsetpagematrix
```

You can use a call to `fmrotatepagematrix` to rotate the page. If you then render text onto that page, a font of zero-degree rotation appears along a rotated baseline. `fmprstr` maintains the current character position, even with rotated text.

Here are a few additional details that can control scaling. Currently, you can scale and rotate only those fonts that have a width vector file. A width vector file has a *.fw* extension. To test that a width vector file is the right one for a particular font, run *strings* on it (as you would for *.ff* family files). All these files are in the default Font Manager directory */usr/lib/fmfonts*.

5.2 The Font Search Path

Although there is a default directory for font files (*/usr/lib/fmfonts*), there is no fixed location for the fonts used by the Font Manager. When the Font Manager needs font data, it searches a path. You can override the Font Manager's default path by setting the environment variable *FONTPATH*. Alternatively, you can use `fmsetpath` to load a new font path. The argument to `fmsetpath` is a colon-separated string of directories.

Because the Font Manager searches a path, you can distribute fonts in different directories. During a font look-up, the Font Manager searches the directories in the order specified (left to right) by the string given to `fmsetpath`. You can use this order to make the Font Manager use a "local" experimental font but still preserve the official font for other users.

For example, if you put the experimental font in your current directory and set FONTPATH to `.:/usr/lib/fmfonts`, the Font Manager uses the font in the current directory, even if that font also exists in */usr/lib/fmfonts*. If the Font Manager fails to find the font in the current directory, the Font Manager searches for the font in */usr/lib/fmfonts*. Thus users can access the "official" font while you modify your local version of the font.

5.3 Late Binding of Fonts

The Font Manager does not create a memory-resident version of a font until it is about to render text onto the screen. Neither `fmfindfont` nor `fmscalefont` put characters in memory (although getting information about a font can cause part or all of the font to be created or read in from disk). The font's residence in memory is controlled by the font cache, through which all character rendering passes.

5.4 Font Manager Routines

The following sections list the Font Manager routines by function. All Font Manager routines are callable from C programs. Table 5-1 gives a summary of each routine.

Task	Routine
`fmcachedisable`	Disable cache flushing
`fmcacheenable`	Enable cache flushing
`fmcachelimit`	Return current cache limit in quanta
`fmconcatpagematrix`	Concatenate page matrix
`fmenumerate`	List font family
`fmfindfont`	Prepare font for manipulation
`fmfontpath`	Get a path for finding fonts
`fmfreefont`	Free memory storage for a font
`fmgetcacheused`	Return total number of bytes used by cache
`fmgetchrwidth`	Return width of a character
`fmgetcomment`	Return a comment associated with a font
`fmgetfontinfo`	Return overall information about font
`fmgetfontname`	Return font's name
`fmgetpagematrix`	Get page matrix
`fmgetstrwidth`	Return width of a string in pixels
`fmgetwholemetrics`	Get information about each character in a font
`fminit`	Initialize the Font Manager
`fminitpagematrix`	Initialize the page matrix to identity
`fmmakefont`	Associate a matrix with a font
`fmoutchar`	Draw a single glyph
`fmprintermatch`	Toggle printer matching
`fmprstr`	Draw a string in the current font
`fmrotatepagematrix`	Rotate the page
`fmscalefont`	Scale a font
`fmscalepagematrix`	Scale the page
`fmsetcachelimit`	Set the maximum cache size in quanta
`fmsetfont`	Set the current font
`fmsetpagematrix`	Set the page matrix
`fmsetpath`	Set a path for finding fonts

Table 5-1. Font Manager Routines

5.4.1 Initializing Fonts

The following routines perform various font initialization and specification functions.

fminit

Call `fminit` to initialize the Font Manager. You must call `fminit` before you can make any other calls to the Font Manager routines.

```
void fminit()
```

fmfindfont

Use `fmfindfont` to get a font handle for a type face. If `fmfindfont` can not find the font, it returns a value of zero. Otherwise, `fmfindfont` returns a handle to a one-point high font of the specified type. The font handle contains information about all the sizes and rotations of that face. `fmfindfont` uses `fmfontpath` to locate the proper directory. To find out which fonts are available, use `fmenumerate`.

```
fmfonthandle fmfindfont(face)
char *face;
```

fmenumerate

Use `fmenumerate` to startup a callback routine for each font face name in the font path. `fmenumerate` uses a string pointer to pass the name of a font face to the callback routine. `fmenumerate` expects the name of the callback routine as an argument.

```
void fmenumerate(clientproc)
void (*clientproc)();
```

For example, the following code uses `fmenumerate` to send font names (via string pointers) to the user-defined routine, `printname`. The `printname` routine prints the name of each font to the terminal:

```
void printname(str)
char *str;
{
        printf("%s\n", str);
}

main()
{
        fminit();
        fmenumerate(printname);
}
```

5.4.2 Sizing Fonts

The following routines control the size of a font.

fmscalefont

`fmscalefont` returns a new font handle. Ideally, the size information in the new font handle is the size specified by *scale*. If you request a size for which there is no font, the Font Manager chooses the closest match available. The other information in new font handle is copied from the font handle passed in by *fh*. Use `fmscalefont` when you want to scale but not rotate a font.

```
fmfonthandle fmscalefont(fh, scale)
fmfonthandle fh;
double scale;
```

fmmakefont

`fmmakefont` returns a new font handle. The transformation matrix passed in by *matrix* is multiplied with the transformation matrix in the font handle passed in by *fh*. This multiplication can scale the font and rotate the baseline. If you want to scale the font but do not want to rotate the baseline, it is easier to use `fmscalefont` than `fmmakefont`. Except for size and rotation information, the information in the new handle is copied from the handle passed in by *fh*. Like `fmscalefont`, if the scaling of a font requests a font that does not exist, the Font Manager substitutes the closest match available.

```
fmfonthandle fmmakefont(fh, matrix)
fmfonthandle *fh;
double matrix[3][2];
```

Note: When using `matrix[3][2]`, think of it as a 2X2 transformation matrix. The last row is reserved for future developement and is currently ignored.

5.4.3 Setting Fonts

The following routine sets the current font.

fmsetfont

Use `fmsetfont` to set the current font (font handle). All subsequent rendering operations use the font handle named by *fh*. To get a font handle, use `fmfindfont` (or `fmscalefont` or `fmmakefont`.)

```
void fmsetfont(fh)
fmfonthandle fh;
```

5.4.4 Rendering Fonts

The following routines render fonts on the screen.

fmprstr

`fmprstr` renders the characters in *str* onto the screen at the current character position. The font used is the one most recently named by `fmsetfont`. The Font Manager starts rendering at the current character position and updates the current character position as it renders. Clients should use the `cmov` and `getcpos` graphics library routines to set or read the current character position.

Before calling `fmprstr`, you must call `cmov` to set the current character position or the results of `fmprstr` are undefined. If the string is NIL, or the font does not exist, `fmprstr` returns –1. Otherwise, `fmprstr` returns zero.

```
    long fmprstr(str)
    char *str;
```

fmoutchar

`fmoutchar` renders a single glyph, *str*, from the current font. If the glyph doesn't exist, the Font Manager advances the current character position by the width of a space. If the font does not define a space character, the Font Manager advances the current character position by the width of the font. The returned value of `fmoutchar` is the width moved.

```
    long fmoutchar(fh, ch)
    fmfonthandle fh;
    unsigned char ch;
```

5.4.5 Getting Font Information

The following routines return information about specified fonts.

fmgetfontname

`fmgetfontname` gets the name of the font associated with the fonthandle in *fh*. `fmgetfontname` writes this information to the location pointed to by *str*. Use *slen* to tell `fmgetfontname` the size of the array pointed to by *str*. `fmgetfontname` does not write more characters than specified by *slen*. If there is an error in locating the font or if no name exists for the font specified by *fh*, the returned value of `fmgetfontname` is –1. Otherwise, the returned value of `fmgetfontname` is the length of the string actually written to *str*.

```
long fmgetfontname(fh, slen, str)
fmfonthandle fh;
long slen;
char *str;
```

fmgetcomment

`fmgetcomment` gets the comment associated with the font handle in *fh*. The comment is written to the location pointed to by the *str* parameter. Use the *slen* parameter to tell `fmgetcomment` the size of the array pointed to by *str*. `fmgetcomment` does not write more characters to *str* than specified by *slen*. If there is an error in locating the font or if no comment exists for the font specified by *fh*, the returned value of `fmgetcomment` is -1. Otherwise, the returned value of `fmgetcomment` is the length of the string written to *str*.

```
long fmgetcomment(fh, slen, str)
fmfonthandle fh;
long slen;
char *str;
```

fmgetfontinfo

`fmgetfontinfo` writes information to the members of the *fmfontinfo* type structure pointed to by the *info* parameter. The information written to this structure pertains to the whole font. The *fmfontinfo* type structure is defined in *<fmclient.h>*.

```
long fmgetfontinfo(fh, info)
fmfonthandle fh;
fmfontinfo *info;
```

The following is a list of possible font information, and what it is used for.

- *printermatched* means there is a printer widths file corresponding to this font.
- *matrix00, matrix01, matrix10, matrix11* are double-precision floats that provide transformation matrix information in points.
- *fixed_width* means all the characters in the font are the same width.
- *xorig, yorig* are the aggregate x-origin and y-origin of the font. *yorig* is the distance from the lowest descender to the baseline. *xorigin* is the distance from the current position to the left edge of the glyph.
- *xsize* and *ysize* are the maximum sizes of the characters in the font, in pixels.
- *height* is often the same as ysize, but some fonts use a larger *ysize* to get free leading (spacing between lines of text).
- *nglyphs* is the index of the highest-numbered character. Indexing begins at 0.

  Note: Some indices may not have glyphs assigned to them, but when you allocate space for `fmgetwholemetrics,` you should use *nglyphs* + 1 as though it were the total number of characters. In other words, *nglyphs* is the highest index of a possibly sparse array.

fmgetwholemetrics

`fmgetwholemetrics` gets glyph information associated with the font handle *fh* and writes it to the *fmglyphinfo* structures pointed to by the elements of the array *fi*. You should allocate enough space to contain *nglyphs*`*sizeof(`*fmglyphinfo*`)`. Because `fmgetwholemetrics` fills only those structures of the array that have corresponding glyphs in the font file, you should initialize all the *fmglyphinfo* structures before calling `fmgetwholemetrics`. (For example, you could use *calloc* to allocate the space. See *malloc*(3C) for more information.)

The returned function value of *fmgetwholemetrics* is 0 if successful. If *fmgetwholemetrics* cannot find the font referenced by the fonthandle, the returned function value is –1.

```
long fmgetwholemetrics(fh, fi)
fmfonthandle fh;
fmglyphinfo *fi;
```

fmgetstrwidth

`fmgetstrwidth` returns the number of pixels the string occupies in the x dimension. It uses the subpixel resolution provided in the glyph widths as it accumulates the width and rounds the sum to the nearest pixel. Rotated fonts are measured along an untransformed x axis.

```
long fmgetstrwidth(fh, str)
fmfonthandle fh;
char *str;
```

fmgetchrwidth

`fmgetchrwidth` returns the number of pixels the given character occupies in the x dimension when rendered. This value is rounded to an integer. If that character glyph does not exist, the width of a space is returned. If a space does not exist, the width of the font is returned. Rotated fonts are measured along an untransformed x axis.

```
long fmgetchrwidth(fh, ch)
fmfonthandle fh;
unsigned char ch;
```

5.4.6 Changing Font Environments

The following routine affects the environment in which fonts are managed.

fmsetpath

`fmsetpath` accepts a pointer to a string that describes the current search path for finding font files. It is a colon-separated list of directories that originate at the root. The default path is "`/usr/lib/fmfonts`".

```
    void fmsetpath(path)
    char *path;
```

fmfontpath

`fmfontpath` returns a pointer to a string that describes the current search path for finding font files. It is a colon-separated list of directories that originate at the root. The default path is "`/usr/lib/fmfonts`".

```
    char *fmfontpath()
```

5.4.7 The Font Cache

The Font Manager tries to restrict its use of memory for fonts to the amount set by the `fmsetcachelimit` command. This command takes a small integer argument that it multiplies by FMCACHE_QUANTUM (see *<fmclient.h>*) to arrive at the maximum number of bytes you want used for font data. The Font Manager allocates memory for fonts when creating printer–matched fonts and rotated fonts. When the amount of memory allocated to these fonts exceeds the limit set by `fmsetcachelimit`, the Font Manager scavenges memory to stay within the bounds of the cache limit.

The cache uses a least-recently-used algorithm to determine what font–assigned memory it should free. Use `fmgetcacheused` to find out the exact number of bytes currently used by font data. `fmcachelimit` returns the current limit in quanta of FMCACHE_QUANTUM.

The routines that control and query the cache follow in alphabetical order. They are followed by a discussion of `fmfreefont`, which should be considered along with caching.

fmcachedisable

The Font Manager provides font cache flushing by default. To disable font cache flushing, call `fmcachedisable`. The Font Manager continues to keep track of the space fonts occupy, but does nothing to limit the size of the space. To restart font cache flushing, call `fmcacheenable`.

```
    void fmcachedisable()
```

fmcacheenable

Use `fmcacheenable` to re–enable the flushing of least-recently-used font–assigned memory that is in excess of the cache limit.

```
    void fmcacheenable()
```

fmcachelimit

The font cache is the memory space the Font Manager uses to store font data. fmcachelimit returns the current maximum size of the font data memory expressed as a multiple of the value FMCACHE_QUANTUM. So, if fmcachelimit returns a 4, and FMCACHE_QUANTUM is 100,000, the cache upper limit size is 400,000 bytes. To set the cache limit, use fmsetcachelimit.

```
    long fmcachelimit()
```

fmgetcacheused

fmgetcacheused returns the exact number of bytes currently allocated to the font cache. There are no implied multipliers. The returned value of this function is the exact number of bytes used.

```
    long fmgetcacheused()
```

fmsetcachelimit

fmsetcachelimit accepts a small integer as an argument, multiplies it by FMCACHE_QUANTUM, and uses the result to reset the upper size limit of the cache data space. If *new-limit* is less than one, fmsetcachelimit resets *new-limit* to one before multiplying by FMCACHE_QUANTUM. If *new_limit* is greater than 100, fmsetcachelimit does nothing.

```
    void fmsetcachelimit(new_limit)
    long new_limit;
```

fmfreefont

`fmfreefont` frees the storage associated with a font in a given rotation and size (as specified in the font handle *fh*). Deleting a font also deletes the font handle. To ensure that `fmfreefont` frees the correct font/rotation/size instance, be sure that the same page matrix is in force as when you first queried or rendered from that font.

Because normal usage of the Font Manager does not involve changing the page matrix, you seldom need to worry about it. But if you find that you cannot delete a font (or have deleted the wrong font) consider the state of the page matrix. An easy way to avoid this problem is to call `fmfreefont` only when the page matrix is not rotated.

As mentioned above, rotated fonts are created and destroyed as necessary and don't need explicit deletion.

```
void fmfreefont(fh)
fmfonthandle fh;
```

5.4.8 Adjusting Widths to Match Laser Printers

Many applications render text on the screen to give the user the chance to proof the text before printing it on a laser printer. For a more realistic simulation, use laser printer character widths to represent the text.

fmprintermatch

`fmprintermatch(0)` disables printer matching, `fmprintermatch(1)` enables printer matching. When the Font Manager renders (images) a font, it inspects the state of this variable. If enabled, the Font Manager searches for a printer widths file that corresponds to the font. If the file exists, and the font has not yet been sized, the Font Manager creates a new font. The Font Manager also updates the font handle of the current font so that it has character widths that correspond to the laser printer's width scheme.

```
void fmprintermatch(set)
long set;
```

5.4.9 Transforming the Page

The page transformation is stated in the page matrix. Use the the following procedures to inspect or change the state of the page matrix.

Note: When using `matrix[3][2]`, think of it as a 2X2 transformation matrix. The last row is reserved for future developement and is currently ignored.

fminitpagematrix

`fminitpagematrix` initializes the page matrix to an orthographic projection.

```
void fminitpagematrix()
```

fmsetpagematrix

`fmsetpagematrix` loads the page matrix verbatim with matrix *mat.*

```
void fmsetpagematrix(mat)
double mat[3][2];
```

fmgetpagematrix

`fmgetpagematrix` returns the page matrix in *mat.*

```
void fmgetpagematrix(mat)
double mat[3][2];
```

fmscalepagematrix

`fmscalepagematrix` uniformly scales the page matrix by *scale.*

```
void fmscalepagematrix(scale)
double scale;
```

fmrotatepagematrix

`fmrotatepagematrix` post-concatenates a rotation to the page matrix. Rotation is measured in a counter-clockwise direction in degrees.

You can also use `fmrotatepagematrix` to generate a screen font that is exactly (within one pixel) the specified size. You should try this in a test program first, to see whether the possible degradation in quality is acceptable. This "roughness" comes from the need to scale a bitmap font if that font does not exist at the specified size.

For example, the Font Manager normally renders text using a bitmap font that is the closest match possible to the requested size. But, if you rotate the page matrix, even by one 1/1000 of a degree, the Font Manager tries to create a font that is rotated that much. As a side effect, the Font Manager also distorts (shrinks or stretches) the page to generate a font that is within a pixel of the specified size. However, stretching or shrinking a bitmap often results in "rough" looking characters.

To try it, call `fmrotatepagematrix(.01)`, then print a string with `fmprstr`.

```
fmrotatepagematrix(angle)
double angle;
```

fmconcatpagematrix

`fmconcatpagematrix` post-concatenates the page matrix with *mat.*

```
void fmconcatpagematrix(mat)
double mat[3][2];
```

5.5 Example

The following example writes a string of green, 25-point characters to a window, beginning at window coordinate (30, 100). Compile the program using the following command line options:

```
cc example.c -o example -lfm -lgl
```

For shared libraries, use the following:

```
cc example.c -o example -lc_s -lfm_s -lgl_s
```

The sample code is below.

```
#include <gl/gl.h>
#include <gl/device.h>
#include <fmclient.h>

main()
{
    short val;
    fmfonthandle font1, font25;

    prefsize(240,210);
    winopen("Hello");
    color(BLACK);
    clear();
    color(GREEN);
    fminit();
    /* Exit if can't find the font family */
    if ((font1=fmfindfont("Times-Roman")) == 0) exit (1);
    /* scale the 1-point-high font to 25 points */
    font25 = fmscalefont(font1, 25.0);
    fmsetfont(font25);
    cmov2i(30, 100);
    fmprstr("Hello World!");

    while(TRUE) { /* redraw window if necessary */
          if (qread(&val) == REDRAW) {
              reshapeviewport();
              color(BLACK);
              clear();
              color(GREEN);
              cmov2i(30, 100);
              fmprstr("Hello World!");
          }
    }
}
```

6. Using the Distributed Graphics Library

The Distributed Graphics Library (DGL) is a facility that allows a process on one machine to use the graphics resources of any IRIS-4D Series workstation on the network. The DGL has two parts:

- a client library linked with application programs that serves as the IRIS Graphics Library interface
- a graphics server, which services requests made by the client graphics program

The client library converts Graphics Library calls into a byte stream. This byte stream is transmitted to the graphics server over the Ethernet or other communication medium. The graphics server program decodes the byte stream and calls the Graphics Library routines to display the graphics on the server's raster subsystem. In this respect, the DGL is a remote procedure call package for the Graphics Library.

The DGL allows IRIS-4D Series workstations to share the work load for graphics applications. A typical application might involve the real time display of an object whose parameters demand a great deal of computation, for example, the simulation of an automobile suspension system. You could use a 4Server node to perform the calculations. Then, acting as a DGL client, display the automobile and its suspension as well as the road surface on an IRIS-4D Series workstation. Both machines can share the work load (each doing the task for which it is best suited) resulting in a more balanced work load and better performance.

6.1 Comparing the DGL with RPC

The DGL is a remote procedure call package for the Graphics Library. It is not a general remote procedure call (RPC) package, and is not a means for general distributed processing. The DGL is designed to split applications that can be partitioned into multiple processes at Graphics Library calls.

There are many differences between the DGL and RPC standards. Most are due to the fact that the DGL is just a Graphics Library RPC and no more. For example, the DGL is non-extensible; it supports only the Graphics Library routines. For non-graphics communication between client and server machines, you must use a separate communication link and a second server process. The DGL offers no facilities for such communications.

Most RPC server processes can talk to multiple RPC clients. The DGL graphics server process talks to only one graphics client. Each graphics client talks to its own unique graphics server process. A graphics server machine can have multiple graphics server processes running simultaneously, each talking to a different graphics client. IRIS workstations can run multiple asynchronous graphics processes. The operating system (not the graphics server) handles the timesharing for the graphics hardware. These design details allow the DGL to devote one server per client, perform authentication only upon initialization, and to make other optimizations to increase performance.

6.2 Installing the DGL

The Distributed Graphics Library (DGL) software comes standard with your workstation and consists of two parts: a client library and a graphics server daemon. Verify that these two software components exist on your workstation: the client library is */usr/lib/libdgl.a* and the graphics server daemon is */usr/etc/dgld*. If either of these files is missing, contact the Silicon Graphics Geometry Hotline for assistance.

6.2.1 DGL Service

The DGL software gets an Internet port number from */etc/services*, see *services*(4). The standard services file has an entry for ''sgi-dgl'' that is commented out. Before using the DGL over the Internet, you must uncomment this entry on both the client and server machines by removing the leading ''#'' character. If you forget to do this, you get error message saying that the service ''sgi-dgl/tcp'' is unknown. The DGL client program prints this error message to **stderr**; the graphics server daemon prints this error message to **stderr** and to the system log maintained by *syslogd*(1M).

6.2.2 *inetd* Configuration

The DGL graphics server daemon for tcp socket connections is automatically started by *inetd*(1M). *inetd* reads its configuration file to determine which server programs correspond to which sockets. The standard configuration file, */usr/etc/inetd.conf*, has an entry for ''sgi-dgl'' that is commented out. Before using the DGL over the Internet, you must first uncomment this entry on the server machine by removing the leading ''#'' character. Then, as superuser, type the following command line to force *inetd* to reread its configuration file:

```
killall 1 inetd
```

If you forget to do this, *inetd* will not start up the graphics server and the DGL client program will time out after about 30 seconds with a connection refused error. To verify that *inetd* is listening, type the following command line and look for a line containing "*.sgi.dgl":

```
/usr/etc/netstat -a
```

6.2.3 *dnserver* Configuration

There are no special DGL modifications necessary beyond the normal 4DDN installation and configuration. The DGL graphics server daemon for 4DDN connections is automatically started by *dnserver*. The file */usr/etc/dn/servers.reg* registers the DGL server with *dnserver* by the name "DGLD". Consult the 4DDN manuals for more information.

6.3 Running Graphics Library Programs with the DGL

Existing Graphics Library programs do not contain any calls that specifically invoke the DGL server. However, these programs can still be run as DGL programs without modifying the source code, simply by relinking them with */usr/lib/libdgl.a* (see Section 6.5.2). The DGL library uses a default set of rules for determining the graphics server, including getting data from the IRIX environment.

6.3.1 Default Connection

The DGL server is initialized by the routine `dglopen`, described in Section 6.4. If `dglopen` is not called by the program, the DGL library attempts to open a default connection by calling `dglopen` with a default server name and connection type. If any of the following environment variables is defined, the server name is the value of the defined variable highest in the following list:

1. *DGLSERVER*
2. *REMOTEHOST*

If the value of *REMOTEHOST* is used for the server name, then the environment variable *REMOTEUSER* is checked. If *REMOTEUSER* is defined, the server name is set to *REMOTEUSER @ REMOTEHOST*. If none of the environment variables above are defined, then the server name is set to the client's hostname.

The value for the connection type comes from the following ordered list:

1. *DGLTYPE* if that environment variable is defined
2. *DGLTSOCKET* if an environment variable is used for the server name
3. *DGLLOCAL*

The environment variable *DGLTYPE* can be set to either the symbolic or numeric value of the connection type, e.g. *DGLLOCAL* or 1.

Note: 4Sight must be running on the graphics server for the connection to be successful. If it is not running, the client will exit with an error.

6.3.2 Using *rlogin*

If you use *rlogin* to remotely log in to an IRIS, *REMOTEUSER* and *REMOTEHOST* are accordingly defined and if *DGLSERVER* is undefined then the DGL program by default establishes a connection back to the last remote machine where you ran *rlogin*. For example, if you *rlogin* from machine A to machine B and then *rlogin* from machine B from machine C, *REMOTEHOST* is set to B on machine C. Therefore, the default graphics connection would be to B.

6.4 Writing DGL Programs

Writing a DGL program is really no different than writing a Graphics Library program, except for optimizing performance. The DGL has all the functionality of the Graphics Library (except where noted in the incompatibilities section below). Applications that require only one graphics server need no modifications. Two new routines for managing server connections have been added for applications that require more than one graphics server.

6.4.1 New Graphics Library Routines

Two new routines, `dglopen` and `dglclose`, have been added to the Graphics Library. These routines allow a DGL program to open and close graphics connections to server machines.

dglopen

`dglopen` opens a DGL connection to a graphics server and makes the new connection the current connection. The first argument specifies the name of the server machine, the second argument the type of connection. If the connection succeeds, `dglopen` returns the *server identifier*, a non-negative integer. Otherwise `dglopen` indicates a failure by returning a negative integer; the absolute value of which indicates the reason for failure.

There are three types of connections supported, *DGLLOCAL* *DGLTSOCKET*, and *DGL4DDN*. *DGLLOCAL* is a local connection, the graphics server is the same machine as the client machine. The first argument, the server name, is ignored for *DGLLOCAL* connections. The graphics server process is forked as a child process and two named pipes are opened for communication. The pipes are created in the directory */tmp* and are unlinked as soon as they are opened. Although the files are unlinked and invisible within the directory, their inodes remain allocated until the pipes are closed. Therefore, the files are guaranteed to be deleted even if the program crashes.

The second type of connection is *DGLTSOCKET*, which is a TCP/IP socket connection. The following sequence of events occurs when a *DGLTSOCKET* connection is attempted:

1. The service ''sgi-dgl'' is looked up in */etc/services* to get a port number. If the service is not found, then an error occurs.

2. The server's name is looked up in */etc/hosts* to get an Internet address. If the host is not found, then an error occurs.

3. An Internet stream socket is created and some of its options are set.

4. A connection to the server machine is attempted with a small timeout. If the connection is refused, the timeout is doubled and the connection retried. If after several tries, the connection is still refused, an error occurs.

5. A successful connection is made and the server's Internet daemon invokes a copy of the DGL graphics server. The graphics server process inherits the socket for communicating with the DGL client program.

6. The graphics server uses *ruserok*(3N) to verify the login. The user id on the server must be the equivalent (in the sense of *rlogin*(1C)) to the user id running the DGL client program or permission is denied.

7. The server process's group and user id are changed according to the entry in */etc/passwd*.

The full format for a server name allows for the specification of many options:

```
[userid [password]@]hostname[#port][:controller[.screen]][;text]
```

userid is the user id for running the graphics server; it defaults to the user id running the DGL client program.

password is optional and is ignored for *DGLTSOCKET* connections. The user id on the server must be equivalent to the originating user id or access is denied.

hostname is the name of the graphics server machine.

port is the port number for establishing a server connection; defaults to the port number for "sgi-dgl" in */etc/services*.

controller is the graphics controller number; it defaults to 0 and is currently ignored.

screen is the screen number; it defaults to 0 and is currently ignored.

text is a comment field and is currently ignored.

The hostname field can be in either of the following formats:

- *ASCII hostname*, translated through */etc/hosts*
- *Internet address*, see *inet*(3N)

If the hostname does not begin with a number, it is assumed to be in the ascii format. Otherwise, it is assumed to be a standard four byte Internet address.

The third type of connection is *DGL4DDN*, which is a DECnet connection. To establish a *DGL4DDN* connection, the correct remote password must be given. The *dnserver* daemon will then start up a process with the correct user and group id. In addition, the server uses *ruserok*(3N) to verify the login just as in *DGLTSOCKET* connections.

If `dglopen` cannot successfully open a connection to a server machine, it returns a negative integer. The absolute value of the returned value is the error number. If a system call or service is the reason for the failure, then the error number returned by the system call (*errno* or *h_errno*) is negated and returned by `dglopen`. Otherwise, the DGL software returns a (negated) value that best indicates the reason for failure. The following table lists the internally generated error values (defined in *<errno.h>*):

Error Value	**Explanation**
ENODEV	invalid `dglopen` type
EACCESS	login incorrect or permission denied
EMFILE	too many dglopen's
EBUSY	only one local gconnection allowed
ENOPROTOOPT	dgl/tcp service not found in */etc/services*
EPROTONOSUPPORT	DGL version mismatch
ERANGE	invalid or unrecognizable number representation
ESRCH	window manager is not running on the graphics server

Table 6-1. `dglopen` Error Values

```
long dglopen(svname,type)
char *svname;
long type

integer*4 dglope(svname,length,type)
character*(*) svname
integer*4 length
integer*4 type
```

dglclose

To destroy a graphics server process and its connection, call `dglclose` with the *server identifier* returned by `dglopen`. This terminates the graphics server process, freeing any system resources that it had allocated, eg. open windows. It also closes the graphics connection and frees any associated system resources on the client machine. Calling `dglclose` with a negative *server identifier* closes all graphics server connections.

After `dglclose` is called there is no current graphics window and no current graphics server connection. Any calls other than `dglopen`, `dglclose` and routines that take graphics window ids as arguments result in an error upon the next communications buffer flush:

```
libdgl error (comm): no current connection
```

Although it is not necessary, it is recommended that `dglclose(-1)` be called before exiting a DGL application. This ensures that the graphics server processes cleanly exit.

```
void dglclose(serverid)
long serverid;

subroutine dglclo(srvrid)
integer*4 srvrid
```

6.4.2 Modified Graphics Library Routines

Two existing Graphics Library routines that were obsolete, `gflush` and `finish`, now take on new functionality. In addition, routines that take graphics window ids as arguments take on additional functionality.

gflush

The DGL client library buffers calls to Graphics Library routines for efficient block transfer to the graphics server. `gflush` explicitly flushes the communication buffers and delivers all buffered, untransmitted graphics data to the graphics server.

Some Graphics Library routines (notably those that return data) implicitly flush the communication buffers. In most programs, the implicit flushing done by routines that return data is sufficient.

The following example outlines a typical use of `gflush`. A program calls some Graphics Library routines that are buffered and not flushed. The program then either computes or blocks for a while, for some non-graphic I/O. `gflush` must be called if the results of the buffered Graphics Library routines need to be seen before and during the pause.

Another reason for using `gflush` is to reduce graphics "jerkiness". If the DGL client is computing data and then sending the data to the graphics server without implicit or explicit flushes, the data will arrive at the graphics server in large batches. The server may process this data very quickly and then wait for the next large batch of data. The rapid processing of Graphics Library routines followed by a pause results in an undesirable "jerky" appearance. In these cases it is probably best to call `gflush` periodically. For example, a logical place to call `gflush` is after every `swapbuffers` call. Be careful not to do too many flushes, either implicit or explicit, as this can adversely effect performance.

```
void gflush()
```

```
subroutine gflush
```

finish

`finish` blocks the client process until all previous routines execute. First, the communication buffers on the client machine are flushed. On the graphics server, all unsent routines are forced down the Geometry Pipeline to the bitplanes. Then, a final token is sent and the client process blocks until the token goes through the pipeline and an acknowledgment has been sent to the graphics server and forwarded to the client process. `finish` is useful when there are large network and pipeline delays.

The following example illustrates a typical use of `finish`. A client calls some Graphics Library routines to display an image. The routines all fit into the server's network buffers and the image takes 30 seconds to render. The client wants to wait until the image is completely displayed on the server's monitor before displaying a message on the client's terminal. `gflush` would flush the buffers but would not wait for the server to process the buffers. `finish` flushes the buffers and waits not only for the server to process all the graphics routines, but for the Geometry Pipeline to finish as well.

```
void finish()
```

```
subroutine finish
```

winopen

`winopen` creates a graphics window on the current graphics server connection and returns a positive integer value identifying the graphics window, or –1 if no additional graphics windows are available. Window identifiers are unique within a DGL program. Each window identifier is composed of the graphics connection identifier (unique within each DGL client program) and the window number (unique within each server process). The window identifier therefore also identifies the window's graphics server connection. See Section 2.2, ''Opening and Closing Windows'' for more information on `winopen`.

Other Window Routines

At any one time, there is only one current graphics server connection. All graphics input and output from the DGL client program goes to or from this server connection. The Graphics Library routines that take graphics window ids as parameters , eg. `winset`, `windepth`, and `winclose`, change the server connection to the connection associated with the window identifier before executing. These routines are the only way to switch graphics I/O back to an existing server connection.

Note that the connection identifier returned by `dglopen` is used only for closing connections and **not** for multiplexing connections. Therefore, when a server connection is created with `dglopen`, a window should be opened and its identifier saved before the graphics server connection changes. Otherwise, there is no way to reconnect to the previous server. See Section 2.5, ''Other Window Routines'' for more information on `winset`.

6.4.3 A Trivial Example

This section shows in a trivial example how to change a standard Graphics Library program into a DGL program. The sample Graphics Library program below clears a window to black and then exits.

```
#include <gl/gl.h>

main ()
{
    winopen ("test");
    color (0);
    clear ();
    gexit ();
    exit(0);
}
```

To change this program into a DGL program that always uses "host1" as the graphics server, add calls to `dglopen` and `dglclose` as follows:

```
#include <gl/gl.h>

main ()
{
    dglopen ("host1",DGLTSOCKET);
    winopen ("test");
    color (BLACK);
    clear ();
    gexit ();
    dglclose (-1);
    exit(0);
}
```

Although this is a very trivial example, the same rules hold for complex programs:

1. call `dglopen` before the very first Graphics Library call
2. call `dglclose` after the very last Graphics Library call

6.4.4 Graphics Library Compatibility

The Graphics Library supports only one local connection to the graphics hardware. To make DGL programs compatible with the Graphics Library, `dglopen` and `dglclose` exist in the Graphics Library but perform little or no function. `dglopen` checks to make sure the type of connection is *DGLLOCAL*. If the type is *DGLLOCAL*, then `dglopen` returns 1, otherwise it returns –ENODEV. `dglclose` is a dummy routine in the Graphics Library and performs no function. DGL programs that open only one *DGLLOCAL* connection will function identically when linked with the Graphics Library.

6.5 Developing Programs

The DGL provides the same routine interface as the Graphics Library and therefore there is little difference in the program development methodology. An existing graphics program can in most cases simply be relinked with the DGL library.

6.5.1 Compiling

Compiling a program for the DGL is no different than compiling for the Graphics Library. In fact, there is no need to recompile source files that have already been compiled into object files. The names for the supported types of connections, eg. *DGLLOCAL* and *DGLTSOCKET*, are defined in the standard Graphics Library header file.

6.5.2 Linking

To link DGL programs, link with */usr/lib/libdgl.a* instead of */usr/lib/libgl.a.* It is also necessary to link with the Sun library. The following command line links the object file foo.o into a DGL executable program:

```
cc -o foo foo.o -ldgl -lsun
```

For languages other than C, link with the wrapper library for the language. The wrapper library should precede the DGL library on the command line. For example, if foo.o in the previous example was a FORTRAN object file then the following command would link the program:

```
cc -o foo foo.o -lfgl -ldgl -lsun
```

Note: The **–Zr** option does not work with the DGL.

6.5.3 Reserved Symbols

A number of prefixes have been reservered for DGL use. These prefixes all end with an underscore character and are not likely to cause conflicts with application programs. The reserved prefixes are listed below for reference:

```
comm_
data_
decnet_
dgl_
gl_
mem_
pipe_
socket_
util_
```

For a complete list of all externally defined symbols, including Graphics Library routines, use the following command:

```
nm -B /usr/lib/libdgl.a | grep " [ABCDEGRST] "
```

6.6 Using Multiple Server Connections

The DGL extends the Graphics Library by supporting connections to multiple graphics servers from one DGL client program. Server processes normally reside on different server machines, but they can also reside on the same machine. The DGL provides mechanisms for creating, multiplexing, and destroying server connections. There are many special considerations to be taken when dealing with multiple graphics servers.

6.6.1 Graphics Input

Each graphics server has its own keyboard, mouse, and optional dial and button box. The graphics input routines, eg. `qtest, qread, getvaluator, setvaluator, qdevice,` and `noise` execute on the current graphics server. The DGL client program can therefore solicit input from multiple keyboards and mice. For most programs, it will make sense to get input from only one graphics server. In all cases, the programmer must make sure that the current graphics server is correctly set when graphics input is solicited.

6.6.2 Local Graphics Data

Each server process runs a separate copy of the Graphics Library and has its own local set of graphics data. For example, linestyles, patterns, fonts, materials, lights, and display list objects are local to each graphics server. When graphics data is defined, it is defined only on the current graphics server; other servers do not define it. One must be careful to reference local graphics data only on the server where it was defined. If a display list or font will be used on multiple servers, then it must be defined on each server.

6.6.3 Possible Applications

There are many applications and advantages to using multiple graphics servers. Here are some of the more common ones.

Some applications may require multiple windows, each with very high resolution graphics. Multiple windows on the same server machine must share one screen's resolution. However with the DGL, an application can control multiple server machines, each of which can devote its full screen resolution to its windows.

Another possible application for multiple servers is for increasing the performance when displaying multiple views of complex objects. If the multiple views are displayed on multiple servers, performance can be linearly increased by the number of servers. For example, an application could create a display list for a car on each of the servers that included material and lighting parameters. Each server could be given a different set of viewing parameters and then told to display the object.

A slight variation of the previous example is to have each server display a different representation of the object. For example, one server could display a depthcued wireframe mesh of the car, another server could display a flat shaded polygonal representation of the car, and another server could display a smooth shaded lighted surface representation of the car. If the display list for each of these representations is very large, multiple servers can eliminate or reduce paging since each server needs only the display list for its representation.

6.7 Limitations and Incompatibilities

The DGL has very few limitations and incompatibilities with the Graphics Library. Below are some of the more important ones.

6.7.1 Limitations

There is currently a limit of one local connection per DGL client program. The current Graphics Library only offers one direct connection to the graphics hardware, so no existing programs will fail due to this limitation.

Each graphics server connection is an open file until the connection is closed. There is a limit of 256 open connections per DGL client program. It is not likely that this limit will ever be reached because the operating system restricts the number of open files per process to a smaller number.

The DGL client and server require enough memory to contiguously store all the arguments to a routine. This is seldom a problem as most routines require very little memory. However, if either a Graphics Library routine requires massive amounts of contiguous memory, or if the system has very little available virtual memory, then performance can be adversely affected or an ''out of memory'' error can occur.

6.7.2 The *callfunc* Routine

`callfunc` is obsolete on the IRIS-4D Series workstations and is provided in the Graphics Library only for compatibility with previous systems. The DGL does not implement `callfunc`. Any references to `callfunc` will result in an undefined symbol error when loading the program.

6.7.3 Pop-up Menu Functions

The DGL server supports a maximum of 16 unique callback functions. Freeing pop-up menus does not free up callback functions. If you use too many callback functions, you get the client error:

```
libdgl error (pup): too many callbacks
```

6.7.4 Interrupts and Jumps

You cannot interrupt the execution of a DGL routine or pop-up menu callback function without returning back to that routine before calling another DGL routine. This can typically happen if you set an alarm or timer interrupt to go off and then block the DGL program with a `qread` call. If the signal handler does not return back to the `qread`, eg. it does a *longjmp*(3C) to some non-local location, unpredictable results are likely.

6.8 Error Messages

All DGL info and error messages are output to the DGL message file. The message file defaults to **stderr**. DGL error messages have the following format:

```
pgm-name error (routine-name): error-text
```

pgm-name is either "libdgl" for client errors or "dgld" for server errors. *routine-name* is the name of the system service or internal routine that failed or detected the error. *error-text* is an explanation of the error.

6.8.1 Client Messages

Client error messages are in the standard DGL error format and are always output to **stderr**. For example, if */etc/hosts* does not include an entry for the server host "foobar", the following error message would be output when a connection to is requested:

```
libdgl error (gethostbyname): can't get name for foobar
```

If the DGL client library detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returned an error number (*errno* or *h_errno*), this number is used as the exit number. The following table lists all exit values that are internally generated (not the result of a failed system call or service).

Exit Value	Explanation
ENOMEM	out of memory
EIO	read or write error

Table 6-2. DGL Client Exit Values

6.8.2 Server Messages

Server error messages are in the standard DGL error format and are output to **stderr** by default. For example, if */etc/hosts* does not include an entry for the client host, the following error messages would be output:

```
dgld error (gethostbyaddr): can't get name for 59000002
dgld error (comm_init): fatal error 1
```

The standard *inetd.conf* file runs the graphics server with the **I** and **M** options. The **I** option informs the graphics server that it was invoked from *inetd* or *dnserver* and enables output of all error messages to the system log file maintained by *syslogd*(1M). The **M** option disables all message output to **stderr**.

If the DGL server is not working properly, check the system log file for error messages. Each entry in the SYSLOG file includes the date and time, identifies the program as ''dgld'' and includes the PID for the server process. The rest of the error message is the text of the error message.

6.8.3 Exit Status

When the DGL graphics server exits, the exit status indicates the reason for the exit. A normal exit has an exit status of zero. A normal exit occurs when either the client calls `dglclose` or when zero bytes are read from the graphics connection. The latter case can occur when the client program exits without calling `dglclose` or abnormally terminates.

A non-zero exit status implies an abnormal exit. If the graphics server program detects a condition that is fatal, it exits with an *errno* value that best indicates the condition. If a system call or service returned an error number (*errno* or *h_errno*), this number is used as the exit number. The following table lists all exit values that are internally generated (not the result of a failed system call or service).

Exit Value	Explanation
0	normal exit
ENODEV	invalid communication connection type
ENOMEM	out of memory
EINVAL	invalid command line argument
ETIMEDOUT	connection timed out
EACCESS	login incorrect or permission denied
EIO	read or write error
ENOENT	invalid Graphics Library routine number
ENOPROTOOPT	dgl/tcp service not found in */etc/services*
ERANGE	invalid or unrecognizable number representation

Table 6-3. DGL Server Exit Values

Appendix A: Textport and Keyboard Data

This appendix describes the escape sequences recognized by *wsh*, and the code sequences generated by the IRIS-4D Series keyboard.

A.1 Escape Sequences Recognized by *wsh*

Table A-2 lists *wsh* escape sequences. The escape sequences use standard ANSI identifiers that are decoded as follows:

	7-bit character code	8-bit character code (hexadecimal)
CSI	ESC [	0x9B
DCS	ESC P	0x90
ST	ESC \	0x9C

Table A-1. ANSI Identifiers for the Escape Sequences

For example, to set the title in the *wsh* title bar to ''Remote host'', the sequence

```
DCS 1 . y Remote host ST
```

is generated by the following C code:

```
printf ("%c%cl.y%s%c%c",'\033','P',"Remote host",'\033','\\');
```

where ESC is the octal 033.

Sequence	Semantics
\012	line feed
\015	carriage return
\007	bell
\010	backspace
\011	horizontal tab
\013	vertical tab (behaves like line feed)
\014	form feed (behaves like line feed)
ESC D	index
ESC E	next line
ESC H	horizontal tab set
ESC M	reverse index
ESC c	reset to initial state (default attributes)
CSI *n* A	move the cursor up *n* lines; sticks at the top
CSI *n* B	move the cursor down *n* lines; sticks at the bottom
CSI *n* C	move the cursor right *n* columns; sticks at the right
CSI *n* D	move the cursor left *n* columns; sticks at the left
CSI H	cursor home
CSI *r* ; *c* H	move the cursor to row *r* and column *c*; these values start at 1
CSI *c* g	clear tab at column *c*
CSI *n* L	insert *n* lines
CSI *n* M	delete *n* lines
CSI *n*	insert *n* characters
CSI *n* P	delete *n* characters
CSI *n* X	erase *n* characters
CSI *code* K	erase in line *code* = 0: erase to end of line from the current cursor column, inclusive *code* = 1: erase from start of line to the current cursor column, inclusive *code* = 2: erase the entire line
CSI *code* J	erase in display *code* = 0: erase to end of display from the current cursor column, inclusive *code* = 1: erase from start of display to the current cursor column, inclusive *code* = 2: erase the entire display

Table A-2. *wsh* Escape Sequences

Sequence	Semantics
CSI *n* {; *n* ...} m	set graphics rendition. Takes 0 or more parameters. if no parameters are given, the graphics rendition is reset to its default. The following are supported rendition values:
	n = 0: reset all modes
	n = 1: bold intensity (implemented as a color change); disables half intensity.
	n = 2: half intensity (implemented as a color change); disables bold intensity.
	n = 4: underline
	n = 5: blinking (recognized but ignored)
	n = 7: reverse video; switches text and page colors
	The following are supported DEC extensions:
	n = 21: clears bold mode
	n = 22: clears half mode
	n = 24: clears underline mode
	n = 25: clears blinking mode
	n = 27: clears reverse video mode
	The following are supported ISO extensions:
	n = 30-37: set the foreground color to *n* - 30
	n = 40-47: sets the background color to *n* - 40
CSI *n* {; *n* ...} h	set mode; no ansi modes are supported
CSI *n* {; *n* ...} l	reset mode; no ansi modes are supported
CSI *r* ; *c* R	cursor position report; recognized but ignored
CSI *n* c	ansi device attributes; recognized but ignored
CSI *n* n	ansi device status report
	n = 5: outputs "CSI 0 n" (CSI zero n)
	n = 6: outputs a cursor report sequence
CSI *r* ; *c* R	where *r* is the cursors current row, starting at 1, and *c* is the cursors current column, starting at 1.
CSI ? *n* n	DEC device status report
	n = 15: outputs "CSI ? 13 n" indicating no printer
	n = 25: outputs "CSI ? 20 n" indicating locked keys
	n = 26: outputs "CSI ? 27 ; 1 n" indicating a north american keyboard type

Table A-2. (continued) *wsh* Escape Sequences

Sequence	Semantics
CSI = *n* {; *n* ...} h	SGI set mode *n* = 6: locks the display where it is for cursor addressing programs; this anchors a *vi* session in place so that the entire retained buffer is not destroyed. *n* == 12: changes into overlay mode
CSI = *n* {; *n* ...} l	SGI clear mode *n* = 6: unlocks the display. *n* = 12: changes out of overlay mode
CSI ? *n* {; *n* ...} h	DEC set mode *n* = 1: enables application keyboard mode *n* = 7: enables auto wrap mode
CSI ? *n* {; *n* ...} l	DEC reset mode *n* = 1: disables application keyboard mode *n* = 7: disables auto wrap mode
CSI *n* {; *n* ...} / y	SGI command *n* = 2: performs textinit *n* = 3: pushes the *wsh* window *n* = 4: pops the wsh window *n* = 101: sets textcolor index to *n* *n* = 102: sets background color (page color) index to *n* *n* = 103: sets bold color index to *n* *n* = 104: sets cursor color index to *n* *n* = 111: sets selection color index; text color = *n*, page color = *n2* *n* = 204: sets window size (in pixels) to width = *n*, h = *n2* *n* = 205: sets window origin (in pixels) to x = *n*, y = *n2*

Table A-2. (continued) *wsh* Escape Sequences

Sequence	Semantics
ESC P *n* {; *n* ...} . y *string* ESC \	SGI device control string *n* = 1: sets the title to *string* *n* = 101: binds a function key to send *string*. If *string* is of zero length, it unbinds the key resetting it to its default value. *string* is an ansi string: no control characters, no characters >= 127. To imbed a control character, use the standard C backslash characters: \r return \n newline \t tab \b backspace Beware of the shell interpreting the backslash (\). *n* = 103 binds a function key locally. *string* = "copy": copies the current selection to the cut buffer *string* = "down-line": scrolls the display down one line *string* = "down-page": scrolls the display down one page *string* = "end": moves the display to the end of the retained buffer *string* = "home": homes the display to the top of the retained buffer *string* = "pop": pops the wsh window *string* = "push": pushes the wsh window *string* = "send": sends the contents of the cut buffer to the program that *wsh* is managing. *string* = "up-line": scrolls the display up one line. *string* = "up-page": scrolls the display up one page.
ESC N	single shift character sets; interprets the next character literally
ESC O	same as ESC N
ESC =	enter DEC keypad application mode; exit DEC keypad numeric mode
ESC >	exit DEC keypad application mode; enter DEC keypad numeric mode
ESC Z	DEC identification (vt52); recognized but ignored
ESC 7	DEC save cursor; recognized but ignored
ESC 8	DEC restore cursor; recognized but ignored
CSI *n* {; *n*} r	DEC set scrolling margins; recognized but ignored
ESC (B	DEC select G0 character set
ESC (0	DEC select G0 graphics set
ESC) B	DEC select G1 character set
ESC) 0	DEC select G1 graphics set
ESC <	DEC enter ansi mode; recognized but ignored

Table A-2. (continued) *wsh* Escape Sequences

A.2 IRIS-4D Series Keyboard Codes

This section describes the code sequences generated by the IRIS-4D Series keyboard. Figure A-1 illustrates the key positions on the 126-key IRIS-4D Series keyboard.

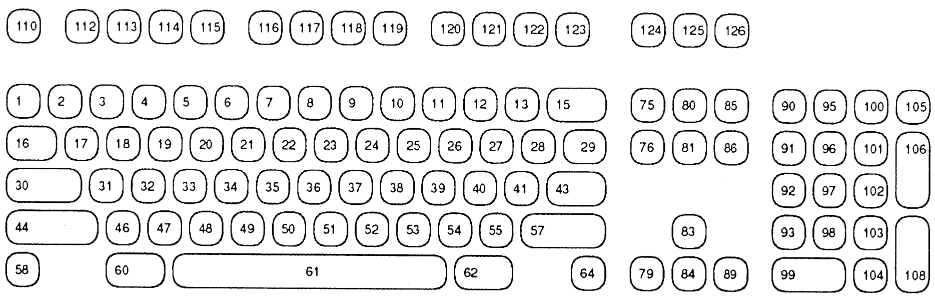


Figure A-1. IRIS-4D Keyboard

The table on the following pages lists each key position and provides the assignment and returned string for each of the four valid key states (base, shift, control, and alternate). These assignments are for the U.S. English version of the keyboard.

key		base	shift		ctrl		alt
1		0x60		0x7e		ESC [0 5 7 q	ESC [1 1 5 q
2	1	0x31	!	0x21		ESC [0 4 9 q	ESC [0 5 8 q
3	2	0x32		0x40	NUL	0x00	ESC [0 5 9 q
4	3	0x33	#	0x23		ESC [0 5 0 q	ESC [0 6 0 q
5	4	0x34	$	0x24		ESC [0 5 1 q	ESC [0 6 1 q
6	5	0x35	%	0x25		ESC [0 5 2 q	ESC [0 6 2 q
7	6	0x36	^	0x5e	RS	0x1e	ESC [0 6 3 q
8	7	0x37	&	0x26		ESC [0 5 3 q	ESC [0 6 4 q
9	8	0x38	*	0x2a		ESC [0 5 4 q	ESC [0 6 5 q
10	9	0x39	(	0x28		ESC [0 5 5 q	ESC [0 6 6 q
11	0	0x30	)	0x29		ESC [0 5 6 q	ESC [0 6 7 q
12	-	0x2d	_	0x5f	US	0x1f	ESC [0 6 8 q
13	=	0x3d	+	0x2b		ESC [0 6 9 q	ESC [0 7 0 q
15	BS	0x08	BS	0x08	DEL	0x7f	ESC [0 7 1 q
16	HT	0x09		ESC [Z		ESC [0 7 2 q	ESC [0 7 3 q
17	q	0x71	Q	0x51	DC1	0x11	ESC [0 7 4 q
18	w	0x77	W	0x57	ETB	0x17	ESC [0 7 5 q
19	e	0x65	E	0x45	ENQ	0x05	ESC [0 7 6 q
20	r	0x72	R	0x52	DC2	0x12	ESC [0 7 7 q
21	t	0x74	T	0x54	DC4	0x14	ESC [0 7 8 q
22	y	0x79	Y	0x59	EM	0x19	ESC [0 7 9 q
23	u	0x75	U	0x55	NAK	0x15	ESC [0 8 0 q
24	i	0x69	I	0x49	HT	0x09	ESC [0 8 1 q
25	o	0x6f	O	0x4f	SI	0x0f	ESC [0 8 2 q
26	p	0x70	P	0x50	DLE	0x10	ESC [0 8 3 q
27	[	0x5b	{	0x7b	ESC	0x1b	ESC [0 8 4 q
28	]	0x5d	}	0x7d	GS	0x1d	ESC [0 8 5 q
29	\	0x5c	|	0x7c	FS	0x1c	ESC [0 8 6 q
30	caps lock	not returned		not returned		not returned	not returned
31	a	0x61	A	0x41	SOH	0x01	ESC [0 8 7 q
32	s	0x73	S	0x53	DC3	0x13	ESC [0 8 8 q
33	d	0x64	D	0x44	CR	0x04	ESC [0 8 9 q
34	f	0x66	F	0x46	ACK	0x06	ESC [0 9 0 q
35	g	0x67	G	0x47	BEL	0x07	ESC [0 9 1 q

Table A-3. IRIS-4D Series Keyboard Codes

key		base		shift		ctrl		alt
36	h	0x68	H	0x48	BS	0x08		ESC [0 9 2 q
37	j	0x6a	J	0x4a	LF	0x0a		ESC [0 9 3 q
38	k	0x6b	K	0x4b	VT	0x0b		ESC [0 9 4 q
39	l	0x6c	L	0x4c	FF	0x0c		ESC [0 9 5 q
40	;	0x3b	:	0x3a		ESC [0 9 6 q		ESC [0 9 7 q
41	'	0x27	"	0x22		ESC [0 9 8 q		ESC [0 9 9 q
43	CR	0x0d	CR	0x0d	CR	0x0d		ESC [1 0 0 q
44	shift	not returned		not returned		not returned		not returned
46	z	0x7a	Z	0x5a	SUB	0x1a		ESC [1 0 1 q
47	x	0x78	X	0x58	CAN	0x18		ESC [1 0 2 q
48	c	0x63	C	0x43	ETX	0x03		ESC [1 0 3 q
49	v	0x76	V	0x56	SYN	0x16		ESC [1 0 4 q
50	b	0x62	B	0x42	STX	0x02		ESC [1 0 5 q
51	n	0x6e	N	0x4e	SO	0x0e		ESC [1 0 6 q
52	m	0x6d	M	0x4d	EOT	0x0d		ESC [1 0 7 q
53	,	0x2c	<	0x3c		ESC [1 0 8 q		ESC [1 0 9 q
54	.	0x2e	>	0x3e		ESC [1 1 0 q		ESC [1 1 1 q
55	/	0x2f	?	0x3f		ESC [1 1 2 q		ESC [1 1 3 q
57	shift	not returned		not returned		not returned		not returned
58	control	not returned		not returned		not returned		not returned
60	alt	not returned		not returned		not returned		not returned
61	SP	0x20	SP	0x20	SP	0x20	SP	0x20
62	alt	not returned		not returned		not returned		not returned
64	control	not returned		not returned		not returned		not returned
75	insert	ESC [1 3 9 q	insert	ESC [1 3 9 q		ESC [1 4 0 q		ESC [1 4 1 q
76	DEL	0x7F	delete	ESC [P		ESC [1 4 2 q		ESC [M
79	←	ESC [D		ESC [1 5 8 q		ESC [1 5 9 q		ESC [1 6 0 q
80	home	ESC [H		ESC [1 4 3 q		ESC [1 4 4 q		ESC [1 4 5 q
81	end	ESC [1 4 6 q		ESC [1 4 7 q		ESC [1 4 8 q		ESC [1 4 9 q
83	↑	ESC [A		ESC [1 6 1 q		ESC [1 6 2 q		ESC [1 6 3 q

Table A-3. (continued) IRIS-4D Keyboard Codes

key		base		shift		ctrl	alt
84	↓	ESC [B		ESC [1 6 4 q		ESC [1 6 5 q	ESC [1 6 6 q
85	page up	ESC [1 5 0 q		ESC [1 5 1 q		ESC [1 5 2 q	ESC [1 5 3 q
86	page down	ESC [1 5 4 q		ESC [1 5 5 q		ESC [1 5 6 q	ESC [1 5 7 q
89	→	ESC [C		ESC [1 6 7 q		ESC [1 6 8 q	ESC [1 6 9 q
90	num lock	not returned	num lock	not returned	DC3	0x13	ESC [1 7 0 q
91	home	ESC [H	7	0x37		ESC [1 7 2 q	0x07
92	←	ESC [D	4	0x34		ESC [1 7 4 q	0x04
93	end	ESC [1 4 6 q	1	0x31		ESC [1 7 6 q	0x01
95	/	0x2f	/	0x2f		ESC [1 7 9 q	ESC [1 8 0 q
96	↑	ESC [A	8	0x38		ESC [1 8 2 q	0x08
97		ESC [0 0 0 q	5	0x35		ESC [1 8 4 q	0x05
98	↓	ESC [B	2	0x32		ESC [1 8 6 q	0x02
99	insert	ESC [1 3 9 q	0	0x30		ESC [1 7 8 q	0x00
100	*	0x2a	*	0x2a		ESC [1 8 7 q	ESC [1 8 8 q
101	page up	ESC [1 5 0 q	9	0x39		ESC [1 9 0 q	0x09
102	→	ESC [C	6	0x36		ESC [1 9 2 q	0x06
103	page down	ESC [1 5 4 q	3	0x33		ESC [1 9 4 q	0x03
104	delete	ESC [P	.	0x2e		ESC [1 9 6 q	ESC [1 9 7 q
105	-	0x2d	-	0x2d		ESC [1 9 8 q	ESC [1 9 9 q
106	+	0x2b	+	0x2b		ESC [2 0 0 q	ESC [2 0 1 q
108	CR	0x0d	CR	0x0d	CR	0x0d	ESC [1 0 0 q
110	ESC	0x1b		ESC [1 2 0 q		ESC [1 2 1 q	ESC [1 2 2 q
112	F1	ESC [0 0 1 q		ESC [0 1 3 q		ESC [0 2 5 q	ESC [0 3 7 q
113	F2	ESC [0 0 2 q		ESC [0 1 4 q		ESC [0 2 6 q	ESC [0 3 8 q
114	F3	ESC [0 0 3 q		ESC [0 1 5 q		ESC [0 2 7 q	ESC [0 3 9 q
115	F4	ESC [0 0 4 q		ESC [0 1 6 q		ESC [0 2 8 q	ESC [0 4 0 q
116	F5	ESC [0 0 5 q		ESC [0 1 7 q		ESC [0 2 9 q	ESC [0 4 1 q
117	F6	ESC [0 0 6 q		ESC [0 1 8 q		ESC [0 3 0 q	ESC [0 4 2 q
118	F7	ESC [0 0 7 q		ESC [0 1 9 q		ESC [0 3 1 q	ESC [0 4 3 q
119	F8	ESC [0 0 8 q		ESC [0 2 0 q		ESC [0 3 2 q	ESC [0 4 4 q
120	F9*	ESC [0 0 9 q		ESC [0 2 1 q		ESC [0 3 3 q	ESC [0 4 5 q
121	F10*	ESC [0 1 0 q		ESC [0 2 2 q		ESC [0 3 4 q	ESC [0 4 6 q
122	F11*	ESC [0 1 1 q		ESC [0 2 3 q		ESC [0 3 5 q	ESC [0 4 7 q
123	F12*	ESC [0 1 2 q		ESC [0 2 4 q		ESC [0 3 6 q	ESC [0 4 8 q
124	print scrn	ESC [2 0 9 q		ESC [2 1 0 q		ESC [2 1 1 q	ESC [2 1 2 q
125	scroll lock	ESC [2 1 3 q		ESC [2 1 4 q		ESC [2 1 5 q	ESC [2 1 6 q
126	pause	ESC [2 1 7 q		ESC [2 1 8 q		BREAK DEL	0x7F

Table A-3. (continued) IRIS-4D Keyboard Codes

* In alternate keypad mode (see Table A-2), keys F9–F12 emulate VT100 keys PF1–PF4.

A.3 Using *wsh* with GL Clients

When you write programs using Graphics Library routines, you can control the size, shape, color, and other attributes of the *textport* to which the program's standard output and standard error are written. A textport is a graphics window that supports text in a way suitable for IRIX input and output. The *wsh* shell window from which the graphics program was started is used as the textport for that graphics program. The Graphics Library allows you to control important attributes of text display using the textport routines, which are explained below.

tpon

`tpon` pops the *wsh* window from which the graphics program was started.

```
void tpon()

subroutine tpon
```

tpoff

`tpoff` pushes the *wsh* window from which the graphics program was started behind all other windows, and effectively hides it.

```
void tpoff()

subroutine tpoff
```

textcolor

`textcolor` selects the color of the text within the *wsh* window from which the graphics program was started.

```
void textcolor(tcolor)
Colorindex tcolor;

subroutine textco(tcolor)
integer*4 tcolor
```

pagecolor

`pagecolor` sets the color of the background of the *wsh* window from which the program was started.

```
void pagecolor(pcolor)
Colorindex pcolor;

subroutine pageco(pcolor)
integer*4 pcolor
```

textinit

`textinit` restores the wsh window from which the program was started to a black background, white text, full size, and central location on the screen.

```
void textinit()

subroutine textin
```

These routines only have an effect when the terminal from which the graphics program has been started identifies itself as a *wsh* window shell. The Graphics Library uses the value of the environment variable *TERM* to determine this, in the following manner:

- if the value of *TERM* starts with "iris", it is assumed to be the *wsh* terminal from which the program is run; otherwise,
- if the name contains, but does not start with "iris", it is presumed to be an IRIS Series 3000 terminal emulator.

For example, if the setting of *TERM* in the C-shell is

```
setenv TERM iris-ansi-66
```

a graphics program started from this shell would attempt to change the text color during each call to `textcolor` because the terminal is identified as a *wsh* window with 66 display lines.

Here is a second example, using the Bourne shell, *sh*:

```
TERM=wsiris; export TERM
```

No action is taken because you have declared the terminal emulator to be an IRIS Series 3000 terminal emulator. Because this terminal emulator is presumed to be remote from an IRIS-4D Series workstation, it makes no sense for the graphics program to change its text color.

wsh has other features that are not available through calls to the Graphics Library. These features are driven by escape sequences that are sent to the textport, just as text can be sent. For example, if the following *printf* statement is included in a C program, the *wsh* title bar will change to 'My Window'.

```
printf("%c%cl.y%s%c%c",'\033','P',"My Window",'\033','\\');
```

For more detailed information regarding escape sequences, see the header file *<gl/ansicode.h>*, or Section A.1 of this Appendix.

Appendix B: Maximizing DGL Performance

This appendix provides detailed information on maximizing the performance of the DGL. This information is intended for use by experienced Graphics Library programmers.

B.1 Analyzing Performance

Estimating expected performance of a DGL application is difficult in the general case. There are many variables and potential bottlenecks to consider: the client host, the communication time and overhead, the server host, and the raster graphics subsystem. Under current circumstances, communication time is usually the bottleneck.

Where data communication is the bottleneck, there are two areas to examine: *one-way traffic* and *turnaround delays*. One-way traffic occurs when the DGL client program sends data to the server without any data being returned. This is usually the most efficient means of data transmission for a communication medium, since many buffering and windowing optimizations can be taken by the low level communication software. Another time when one-way traffic occurs is when the server returns large quantities of data to the client. This is an infrequent operation, especially for amounts of data larger than 1000 bytes. For less than 1000 bytes of returned data, the turnaround delay described below is the limiting factor.

Turnaround delays occur when the client sends data to the server and then waits for data to be returned. The server first must read all buffered data, interpret it, and then send a reply. The client has to wait for this reply to be received and must then interpret it as well. Inter-machine turnaround delays are usually quite long. Turnaround delays on the local machine can be reasonably short. Turnaround delays occur for any routines that return data, such as `qtest`, `getvaluator`, `getmatrix`, `readpixels`, `endfeedback`, and `finish`.

Below is a benchmark program that will allow you to calculate the maximum performance levels (one-way kbytes and turnarounds per second) for your IRIS workstation over the communication options currently available.

```
/*
USAGE

bench [hostname]

If hostname is specified, then a DGLTSOCKET connection is opened
to hostname.  Otherwise, dglopen is not called. To benchmark
a DGLLOCAL connection, first "setenv DGLTYPE DGLLOCAL" then run
bench without any arguments.  Output is self-explanatory.
*/

#include <stdio.h>
#include <sys/time.h>
#include <gl/gl.h>

main (argc,argv)
    int argc;
    char **argv;
{
    register long i,count,start,end;
    int its;
    float secs;
    static struct timeval tpstart,tpend;

    count = 20000;
    if (argc == 2) dglopen(argv[1], DGLTSOCKET);
    else if (argc != 1) usage();

    noport();
    foreground();
    winopen("");
    finish();

    its = 100000/count;    /* benchmark bbox2i in immediate mode */
    if (its < 1) its = 1;  /* since it is basically a nop */
    while (its-- > 0) {
        fprintf(stderr,
               "Sending %d bbox2i() commands (24 bytes each) ... ",
               count);
        fflush(stderr);
        gettimeofday(&tpstart,NULL);
        for (i = 0; i < count; i+= 4) {
            bbox2i(1,2,3,4,5,6);
            bbox2i(1,2,3,4,5,6);
            bbox2i(1,2,3,4,5,6);
            bbox2i(1,2,3,4,5,6);
        }
        finish();
        gettimeofday(&tpend,NULL);
        secs = (tpend.tv_sec - tpstart.tv_sec) +
                1e-6 * (tpend.tv_usec - tpstart.tv_usec);
        fprintf(stderr,"done.\n%.2f seconds, %d bytes per second\n",
                secs,(int)(count*24/secs));
    }

    count /= 10;             /* benchmark getpattern - it is very fast */
    if (count < 1) count = 1;  /* therefore its speed is not an issue */
    its = 10000/count;
    if (its < 1) its = 1;
    while (its-- > 0) {
        fprintf(stderr,
        "Sending %d () getpattern commands (1 turnaround each) ... ",
            count);
        fflush(stderr);
        gettimeofday(&tpstart,NULL);
        for (i = 0; i < count; i+= 4) {
            getpattern();
            getpattern();
            getpattern();
            getpattern();
        }
        gettimeofday(&tpend,NULL);
        secs = (tpend.tv_sec - tpstart.tv_sec) +
            1e-6 * (tpend.tv_usec - tpstart.tv_usec);
        fprintf(stderr,"done.\n%.2f seconds, %d turnarounds per second\n",
                secs,(int)(count/secs));
    }

    gexit ();
    dglclose (-1);
}

usage ()
{
    fprintf(stderr, "Usage: bench [remotehost]\n");
    exit(1);
}
```

One-way traffic rates assume that flushes and/or turnarounds do not happen frequently; frequent flushes and/or turnarounds can substantially reduce the quoted rates. Turnaround performance can be a factor of 10 times slower under certain worst-case conditions.

When you are computing data bytes, note that each Graphics Library routine requires a longword (4 byte) header in addition to the argument data to be transmitted, e.g., `movei(1,2,3)` requires 16 bytes and `draw2s(5,6)` requires 8 bytes. The DGL aligns data according to the MIPS conventions and adds padding if necessary, eg. `color(0)` requires 8 bytes because the next routine must be longword aligned.

You can estimate a DGL application's maximum performance by determining how much communication traffic of each type occurs within the application. This calculation yields a fairly accurate estimate of performance assuming the application is communications limited. The DGL has built-in monitoring capabilities that can tell you how much data is communicated and how many turnarounds occurred.

B.2 Monitoring Communication Traffic

Before you can monitor communication traffic you must take a few preliminary steps. The first is to establish a benchmark. This benchmark should have no user or external interaction. This is to ensure that the benchmark measures computation and doesn't measure waiting time for external input. If possible, you should specify a preferred window position within the program (using `prefposition`) to avoid the need to manually specify a window's location. The benchmark should also be as graphics intensive as possible. Non-graphics code can be benchmarked and optimized using *prof* and *pixie*. The benchmark should be longer than 10 seconds of real time. Opening and closing a connection and a window may take up to 1 second of real time. By making the benchmark longer than 10 seconds, the variable initialization time is less than 10 percent.

The second step is to call `dglclose(-1)` after `gexit`. If debugging is on, `dglclose` displays performance data to **stderr**. To turn on debugging within a DGL client program, set the debugging level to 1 by typing the following command line:

```
setenv DGLDEBUG 1
```

When the DGL library opens a connection to a server, this environment variable is translated and its numeric value becomes the debugging level.

The client program will output something similar to the following to **stderr**:

```
libdgl info.1: comm_default_init (server=user@host, type=2)
libdgl info.1: dgl_open (user@host#-1, 2) from client user@host2
libdgl info.1: creating stream socket..setting..connecting..done.
libdgl info.1: client is 'host2' on port 1212
libdgl info.1: server is 'host' on port 5232
libdgl info.1: number of client errors: 0
libdgl info.1: number of server errors: 0
libdgl info.1:
Elapsed and CPU time
--------------------
real    28.54
user     0.17
sys      1.59

Communication statistics
------------------------
        ---write---                          ---read---
reallocs      0 (0.00/sec)                   0 (0.00/sec)
dglbufs    5507 (192.96/sec)              5506 (192.92/sec)
nexbufs       0 (0.00/sec)                   0 (0.00/sec)
nios       5507 (192.96/sec)             11012 (385.84/sec)
nbytes   114256 (4003.36/sec)            96056 (3365.66/sec)
turnarounds = 5506 (192.92/sec)
Final comm_bufsize = 4096
```

After the first several informational messages, the program's elapsed time and CPU time are displayed. The DGL library begins measuring time as soon as the first server connection is opened and stops measuring when `dglclose` is called. There are two important numbers within the communication statistics. The first is the total number of bytes read and written. This number is the sum of the two columns on the line that begins with ''nbytes'', in this example 114256 plus 96056. The other number is the number of turnarounds which is 5506.

Now that you know the number of bytes transmitted and the number of turnarounds between the client and server, you can determine if the communication medium is the bottleneck. Compute the estimated time for the application by computing the sum of the one-way and turnaround traffic times. The one-way traffic time is the number of bytes read and written divided by the number of bytes/second for the medium. The turnaround time is the number of turnarounds divided by the number of turnarounds/second for the medium. If the total time in seconds is within 1 second of the total elapsed time for the application, then the communication medium is the bottleneck. To double check this conclusion, verify that the real time is more than the sum of the user and system times.

In this example, if we had benchmarked the rate of one-way traffic at 280 kbytes/sec and the turnaround rate at 200/sec, the one-way traffic time would be 210312/280000 or .75 seconds and the turnaround traffic time would be 5506/200 or 27.5 seconds. The total traffic time is 28.25 seconds compared to 28.54 elapsed time. Therefore, the communication medium is the bottleneck, due to turnaround delays. The fact that the user time of 0.17 plus the system time of 1.59 is much less than the real time reinforces the conclusion that the client program is not the bottleneck. Also notice the ratio of system time to user time. The application program and DGL code for encoding and buffering graphics data is user code. The sys time is the time spent reading and writing the communication medium. In this example, the ratio of system/user time is about 10. This indicates that the communication medium is definitely the bottleneck. The lower the ratio, the more balanced the application and communication medium are. Note that this assumes that the application program does not account for any system time itself.

If the traffic time is substantially less than the real time then communication is not always the limiting factor. If the user time plus the system time for the client program is more than half the real time, then the client program might be compute bound at times.

B.3 Locating Bottlenecks

In complex applications, the bottleneck will not be in just one part of the system. Rather, each part of the system may be a bottleneck for some part of the time. For example, the bottleneck may be the client program while computing a surface, the communication medium while transferring the surface to the server, and then the server while clearing the screen and rendering the surface.

B.3.1 Communications Bottlenecks

If the application program is running at the maximum estimated rate that the communication medium can support, then the communication medium is the bottleneck. Often, however, the actual communication rates are much less than the maximum. Therefore, the communication medium can still be the bottleneck even though the program's performance is less than the maximum allowed by the communications. There are two methods to verify if the communication medium is the bottleneck.

The first method is to run the application locally. The *DGLLOCAL* communication type has the highest performance of all communications media. If performance substantially increases due to the increase in communications speed, then communications is the bottleneck. The percentage increase in performance indicates how limiting communications is for the application.

The second method is to determine if both the client and server are idle when the benchmark is running. If both machines have substantial amounts of idle time then the communications medium is the bottleneck. To determine idle time, run *sar(1M)* on each machine by typing:

```
sar 5 100
```

This outputs one line of status every 5 seconds, 100 times. Adjust the time interval and iterations as necessary. Check the ''%idle'' column to see how much of the time the system is idle. If both machines show substantial idle time, then the communications medium must be the bottleneck. Make sure that the benchmark is the only program running.

If the communication medium is not the bottleneck, then either the client and/or the server is the bottleneck. The best way to look for bottlenecks is to start at the raster subsystem and work backwards through the graphics server to the client program.

B.3.2 Raster Subsystem Bottlenecks

The raster subsystem can be the bottleneck for graphics routines that take a long time to process, eg. screen clears and large polygon fills. Run *sar* on the graphics server to see if the graphics pipeline is blocking the server. The ''%wio'' column is the percent of time the system is waiting for I/O. The ''%wfif'' column is the percent of the wio time that is spent waiting because the graphics FIFO is too full. If the ''%wfif'' column is more than zero then the raster subsystem is a bottleneck some of the time. Multiply the ''%wfif'' by the ''%wio'' column to find the percentage of time the system is waiting for the raster subsystem.

B.3.3 Graphics Server Bottlenecks

Check the ''%idle'' column to see how much of the time the graphics server is idle. If the server is idle more than 30% of the time then the server is not the bottleneck. The server is usually blocked on input. If this is true, the bottleneck is most likely in the client program or the communication medium.

If the ''%idle'' column is less than 30%, then the graphics server is probably a bottleneck some of the time. With relatively slow communication options, the only way the server can be the bottleneck is if a few bytes of data cause the server to perform a large amount of processing. Another way to state this as follows: the larger the ratio of server CPU processing to data bytes, the more likely it is that the server process is the bottleneck. There are three types of operations that have large ratios:

display lists	Calling a display list requires only 8 bytes of data transmission. Depending on the size and hierarchy of the display list, the server process can spend an unbounded amount of time traversing and executing the dislay list.

curves and surfaces	Tesselation of curves and surfaces is done either in the CPU or in graphics hardware, depending on the type of raster subsystem. If the tesselation is done in the CPU, then the server process will spend a large amount of CPU time tesselating.
object editing	Editing display lists can cause memory fragmentation. This in turn can cause the object editing code to become inefficient. Although this is unlikely to occur, it is still a possibility.

In any case, if the graphics server is the bottleneck, then *prof* should be able to pinpoint the reason why. By using *prof* and *pixie*, you can determine which Graphics Library routines were called, from where they were called, how many times they were called, and how much time they took.

B.3.4 Client Program Bottlenecks

To determine if the client program is a bottleneck, run *sar* on the client machine. Check the ''%idle'' column to see how much of the time the system is idle. The ''%idle'' number should be equal to 100 percent minus the user time plus the system time divided by the real time for client program. You can expect turnaround traffic to consume 10% (at most) of the client machine and one-way traffic to consume 50-60% (at most) of the client machine.

If the system is less than 40% idle then the client program is probably compute bound at times. If the machine is hardly ever idle or usually less than 10% idle, then the client program is definitely a bottleneck. There can be many reasons for this, such as paging, disk or tape I/O, or excessive computation. The output from *sar* tells you where the CPU time is spent. If most of the time is spent executing user code, then *prof* and *pixie* can help analyze the user code.

The DGL code that translates Graphics Library routines into a buffered data stream is very efficient and is not likely to be the bottleneck. Therefore, most client program optimizations will be in application code. However, proper use of display lists can help reduce the number of Graphics Library calls and DGL overhead.

B.3.5 Reducing Communication Traffic

If data communication is the bottleneck, then the simplest way to improve performance is to reduce the amount of data communication. Even if communications is not the bottleneck, it is still advisable to minimize communication traffic. Any improvement in communications will still improve overall performance.

There are specific methods to reduce the two types of communication traffic. The next section deals with one-way traffic and how to reduce it to improve peformance. The final section deals with turnaround traffic and how to optimize performance.

B.3.6 Handling One-way Traffic with Display Lists

The best way to reduce one-way traffic is to use display lists. On a workstation with integrated graphics, display lists are often more trouble then they are worth. For most applications, displays lists represent yet another copy of the design database. This extra representation has two main disadvantages: memory space and consistency. A display list of a design database will often occupy as much memory space as the database itself. Whenever the database is updated, the display list needs updating to keep it consistent, often requiring the use of tags and object editing. This is turn can lead to memory fragmentation and further increase memory usage.

However, there are many differences in the DGL environment of separate client and server machines. When the client and server are different machines, communications is often the bottleneck. Both the client and server have their own memory; using the server's memory to store a duplicate copy of the client's graphics database is not wasteful. Rather, it is a wise use of memory to save communication time. In many DGL applications, proper use of display lists is the easiest and best way to improve performance.

A simple rule of thumb for the DGL is that it takes almost no time to call a display list, and only a little longer to create a display list than it does to perform the graphics in immediate mode. Therefore, if the object is to be drawn more than once, it is extremely advantageous to build a display list first, and then call the display list multiple times. The more times the display list is called, the more the savings. However, the advantage diminishes with increased one-way traffic performance.

The consistency problem still remains. Whenever the client program changes its database, it still needs to change the display list on the graphics server. There are two methods of doing this. One method is to simply redefine the entire display list. This works very well for smaller display lists or when the design can be partitioned into small display lists. The other method is to use the object editing routines to edit the display list and change it. This works well for very large display lists where the cost of editing is less than the cost of redefining the display list. Each application has to weigh the advantages and disadvantages of each method before chosing the more appropriate method or combination of methods.

IRIS-4D Series workstations support a robust lighting model along with hidden surface removal. It is possible to create a static display list that contains all the information necessary to view an object from any position and angle, under any lighting conditions. If the viewing transformation and lighting parameters are kept separate from the displayed object, then the object can be moved about and rotated, under varying lighting conditions, without ever editing the object's display list. Minor attribute changes, such as material color or reflectivity, can easily and efficiently be changed using display list editing. Therefore, very large databases can be efficiently stored in a display list on a graphics server with a client program controlling the viewing parameters.

B.3.7 Handling Turnaround Delays

If too much time is spent waiting for turnaround delays, there are two alternatives. One alternative is to try to reduce the need for so many turnarounds. For example, queuing a valuator with a reasonable noise level is much more efficient than polling the valuator. In general, polling is not recommended on shared communication media because it places a constant load on the shared network.

The second alternative is to use `getdev` and `blkqread`. These routines were specifically designed for reducing turnaround delay. `getdev` polls and returns the values of multiple valuators. `blkqread` reads and returns multiple queue entries. Both require only one turnaround time and make very efficient use of the communication medium.

Index

Section 2:

Programming in NeWS

Contents

List of Tables

1. Introduction

The Network-extensible Window System (NeWS) is a distributed, extensible window system developed by Sun Microsystems. This chapter provides an overview of the NeWS model of screen interaction.

1.1 NeWS Design Basis

NeWS is based on a novel form of interprocess communication. Processes usually communicate with each other by exchanging messages via some communication medium. These messages usually consist of streams of commands and parameters. A NeWS process, however, communicates by sending entire programs that are then executed by the receiving processes.

PostScript was originally designed to enable people to communicate with a printer. To print on a PostScript-based printer, you transmit PostScript programs to the printer. A processor in the printer receives and executes the programs, which in turn causes an image to appear on the page. The ability to define a function in the PostScript language allows the extension and alteration of the capabilities of the printer.

PostScript is the language used by NeWS. NeWS is structured as a single IRIX process, which acts as a network server and contains a PostScript interpreter. Within this server process is a collection of *lightweight processes* that execute PostScript programs. A lightweight process can be thought of as a thread of control for executing a PostScript program. Client programs talk to NeWS through byte streams. Each of these streams has a lightweight PostScript process within the NeWS process that executes it.

Messages pass between client processes, which can exist anywhere on the network, and PostScript processes, which exist within the NeWS server. These processes can perform operations on the display and receive events from the keyboard and the mouse. They can also talk to other PostScript processes.

NeWS is centered around PostScript as an extension language. All that is provided by NeWS is a set of mechanisms; policies are implemented as PostScript procedures. For example, NeWS has no window placement policy. It has mechanisms for creating windows and placing them on the screen, given coordinates for the window. The choice of those coordinates is up to a separate PostScript procedure.

What is usually thought of as the user interface of a window system is explicitly outside the design of NeWS. User interface includes such things as how menu title bars are drawn and whether or not the user can stretch a window by clicking the left button in the upper right hand corner of the window outline. All these issues are addressed by implementing appropriate procedures in PostScript.

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

1.2 Imaging

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

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

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

Rather than use pixels as units of measure on the screen, NeWS uses *points*. There are 72 points to the inch, thus making the screen area 960 points wide by 768 points high. To convert from pixel measure to point measure, multiply all pixel values by 0.75.

1.3 Canvases

NeWS's basic drawing surface is a *canvas*. This term was picked to avoid the semantic confusion that surrounds the word ''window.'' A canvas is just a surface on which an image may be drawn. The surface may be either opaque or transparent, and can have an arbitrary shape, and can overlap.

Canvases are easy to create. Menus, windows, and pop-up messages are all based on canvases. PostScript has been extended with primitives to create and manipulate canvases. All PostScript graphics operations are performed on some canvas.

1.4 User Interaction

Each possible input action is an *event.* Events are a general notion that includes buttons being pressed (buttons may be on keyboards, mice, tablets, or whatever else) and locator motion. They are implemented as messages between PostScript processes.

Events are distinguished by where they occur, what happened, and to what. The objects are usually physical; they are the things that a person can manipulate. An example of an event is a key being pressed while the mouse is over a canvas. This might trigger the transmission of the ASCII code for the character pressed to the process that created the canvas. The bindings between events and actions are very loose and easy to change.

The actions taken when an event occurs can be specified in a general way, via PostScript. The striking of the key sends a message to a PostScript process that decides what to do with it. The process can do something as simple as sending the message to a IRIX process, or as complicated as inserting the message into a locally maintained document.

PostScript procedures control much more than just the interpretation of keystrokes. They can be involved in cursor tracking, constructing borders around windows, doing window layout, and implementing menus.

1.5 Client Interface

A client program exists in either one or two parts: the first part is written in PostScript and lives inside NeWS, and the second (optional) part lives outside NeWS and talks to it through a byte stream. Thus, you can view the client interface at three levels of sophistication.

- At the first level, the programmer writes PostScript programs and deals with an entirely PostScript universe. Menu packages and window layout policies are examples of objects that will usually be implemented this way.

  At this level, NeWS provides conventions that define an object-oriented interface to windows, menus, selections, and so on. Objects inherit a default set of behaviors, but these can be overridden selectively. You can write a NeWS client entirely in PostScript at this level.

- At the second level, the programmer writes programs in C, or some other language, that generate PostScript programs. The programmer is explicitly aware of the existence of PostScript. NeWS emulators for other window systems are generally implemented this way. The *roundclock* clock program is another example of a program written using this method.

  At this level, NeWS provides a C pre-processor that allows C programmers to cross the language boundary to PostScript easily. In effect, they can write C procedures with PostScript bodies.

- At the third level, the existence of PostScript and message passing is completely hidden by an interface that someone has constructed using the second level facilities. The GL interface is an example of this approach.

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

2. Interacting with NeWS

This chapter explains how to get started with NeWS and PostScript programming, including some simple ways to "customize" your NeWS server.

The best way to learn NeWS is to read the two PostScript manuals, *The PostScript Language Reference Manual* and *The PostScript Language Tutorial and Cookbook.* As you read these, try out the example PostScript programs from these books using the PostScript shell, *psh,* described below. You should insert the **pause** operator into any loops (**for, loop, repeat**) in the PostScript manuals' examples. Once you feel comfortable with the PostScript language, you should read Chapters 3-6, which describe NeWS extensions to PostScript. As you do this, you can start examining and modifying the demo programs supplied with NeWS in */usr/NeWS/clientsrc.* At the same time, you can try adding or changing features in the window manager via the *user.ps* file.

2.1 Using the PostScript Shell

Central to the NeWS development environment is *psh*(1), the NeWS PostScript shell, which establishes a direct, interactive connection to the NeWS server. The *psh* command provides an easy way to send PostScript programs to the NeWS server. *psh* establishes a TCP/IP connection to the NeWS server and sends it PostScript code from files or standard input. If your program can run directly in the NeWS server, (i.e. it is written entirely in PostScript), you can use *psh* to execute the program as a script.

2.1.1 Starting the PostScript Shell

The NeWS server is a PostScript interpreter, and you can use it to interactively program and debug. To start *psh*, follow these steps:

1. At the system prompt, enter the *psh* command.

   ```
   % psh
   ```

   You are now in the NeWS PostScript shell. This shell does not use a prompt; enter subsequent commands when the cursor returns to a new (empty) line.

2. Invoke a PostScript executive process. The NeWS server responds with a greeting.

   ```
   executive
   Welcome to 4Sight Version 1.4
   ```

Now any PostScript commands you type will be executed on the IRIS screen.

To quit *psh*, type the following on a new line:

```
bye
%
```

This will return you to your normal system prompt.

2.1.2 *psh* Scripts

As in any other shell, you can create scripts that can be run as executable files. To make an executable PostScript file, you must include the following first line:

```
#! /usr/NeWS/bin/psh
```

You must also also make sure the file is executable, using *chmod*(1):

```
% chmod +x file.ps
```

Now when you type the name of the file at the system prompt, *psh* will automatically be invoked, and the program will execute on the IRIS screen.

2.1.3 Previewing PostScript Graphics

You can use *psh* to directly preview PostScript Graphics by creating your own window and defining a **PaintClient** procedure for it that includes your PostScript code. **PaintClient** is called whenever the window is resized or damaged.

If you do not define a window in *psh*, and execute raw PostScript in it, the results will be drawn on the background, but will not be repainted if damaged.

You should also be aware of some differences between *psh* and a PostScript printer. First, various printer-related commands such as **showpage** are meaningless or undefined in the NeWS window environment. Second, not all PostScript primitives have been implemented in the current version of NeWS (see Appendix A, ''Implementation Notes'').

The program *psview*(1) implements a simple page previewer that can be used to preview many files that can be sent to a PostScript printer.

2.2 Displaying Messages with *say*

To display a message in a window, use the *say*(1) program. This program has many options, but its default action is to create a message in a window. The following example creates a window with the frame header ''Text Using Say'' and text within the window which reads ''Hello There''.

```
% say -b"Text Using Say" Hello There
```

You can also use *say* to display raw PostScript. *say* has an advantage over *psh*, because *say* automatically creates a window for you. See the *say*(1) manual page for information on the various options available with this program.

2.3 Using NeWS over the Network

NeWS is a network-based window system, so you can connect to remote NeWS servers and display output on them (remember, the *server* runs on the machine with the display and keyboard, providing them as a resource for the *client* program).

2.3.1 Connecting to Remote NeWS Servers

The environment variable *NEWSSERVER* determines which server client programs will access; by default they access the local host. There is a utility program, *setnewshost*(1) which outputs the correct setting of the *NEWSSERVER* variable for a given remote host.

After you define *NEWSSERVER*, *say* and other NeWS client programs will display their output on *remote_host*, and *psh* will connect to *remote_host* allowing you to run an interactive programming session on the remote machine. You can redefine *NEWSSERVER* with the following command line:

```
% setenv NEWSSERVER `setnewshost remote_host`
```

You can also use the network aspect of NeWS to run NeWS or DGL applications on remote machines while you interact with them on your own workstation.

Note: See Section 1 of this manual, ''Using the GL/DGL Interfaces'', for information on how to display IRIS Graphics Library programs on remote machines.

2.3.2 Reaching NeWS from a Remote System

If you are running on a remote system that does not have *psh*, you can still connect to the NeWS server by using *telnet*(1) to access IP port **2000** on the workstation running NeWS.

```
% telnet hostname 2000
```

telnet will not echo your characters when you type **executive.**

2.4 Customizing NeWS with *user.ps*

When you start up the NeWS server, (i.e., log in from the console window) it first executes the PostScript file *init.ps*, which loads and executes a standard set of PostScript files that define the classes, packages, and user interface for the window manager. See Section 3 of this manual, ''Programming the IRIS Window Manager'', for the names and contents of these files.

Note: Whenever the server opens a file (including *init.ps*) it first tries to open the file in the current directory, and if this fails, it looks for it in */usr/NeWS/lib*.

After this, *init.ps* looks for another PostScript file named *user.ps*. The NeWS server then executes this file as well. *user.ps* is supported as a means for you to ''customize'' the behavior of the server. A default version of *user.ps* can be found in */usr/NeWS/lib*. You should copy it to your home directory, and make any changes to it there.

2.4.1 Creating *user.ps*

NeWS supports the *user.ps* file as a convenient place for you to override default settings in the standard *init.ps* file, replace definitions or procedures from the other startup files, and define useful procedures for your own use. You can change any window manager function defined in the initialization files simply by placing your own version of the function in a file in your home directory with the name *user.ps*.

Once you have created and edited your *user.ps* file, you must log out of 4Sight (choose 'Log Out' from the Max menu after saving and/or quitting from any applications or shells). When you log back in, *init.ps* will load the changes you made in your *user.ps* file into the NeWS server. See Section 3 of this manual, ''Programming the IRIS Window Manager'' for more complete descriptions of what can be put in your *user.ps* file, as well as other ways customize your windowing environment.

2.4.2 Recovering From Errors

If there is a syntax or logic error in the PostScript code you put in *user.ps*, the 4Sight server may abort. When this happens, you are automatically logged out. To recover from this problem, log in as follows:

```
login_name   NOGRAPHICS=1
```

You will now be logged in, but the 4Sight server will not be started. You will not be able to use any visual editor (like *vi*) from the console window, so the best idea is to move *user.ps* to another file, which you can edit after you start up the default version of the server. Type the following:

```
mv user.ps user.nogood
```

Now, logout from the console window, and then log back in as usual. The default version of the 4Sight server will start up, and you can edit *user.nogood* and move it back to *user.ps* when you are done.

To help you debug, you may want to check the end of the file */usr/adm/SYSLOG*, which records any error messages generated by the 4Sight server. Type the following to view the tail end of */usr/adm/SYSLOG*:

```
tail /usr/adm/SYSLOG
```

2.4.3 Expanding the User Dictionary

When client programs are run and first start defining procedures, they make entries in a per-process user dictionary. However, the *user.ps* file adds procedures to the system dictionary, which has a finite amount of room available. More importantly, the system dictionary is shared by all processes, so you do not want to clutter it up with lots of definitions because this will increase the risk of name clashes.

If you are going to define lots of new functions, it's best to create your own dictionary from within *user.ps* as follows :

```
% Define my VDI emulation routines
systemdict /myVDIdict known not {
    systemdict /myVDIdict 50 dict put
    myVDIdict begin
    /VDIrange 34200 def
    etc.
    end
} if
```

This checks to see if '**VDIdict**' is already defined in the system dictionary; if it isn't, it creates its own dictionary, only adding one entry to the system dictionary. You can access your own dictionary of extensions as follows:

```
myVDIdict begin
VDIrange 4 mul
etc.
end
```

(Or you can use the **get, store** and **put** primitives.)

The following two sections contain examples of modifications you can make in *user.ps.*

2.4.4 Changing Your Background Pattern

Placing the following code in *user.ps* will change the background pattern to grey with a dark red border. After editing your *user.ps* file, you must choose 'Log Out' from the Max (background) menu; when you log back in, *user.ps* will load the new background.

```
/PaintRoot {
    gsave framebuffer setcanvas
    .4 setgray clippath fill
    .5 0 0 setrgbcolor
    20 20 925 738 rectpath closepath
    3 setlinewidth
    1 setlinequality
    stroke
    grestore
} dup /PlainPaintRoot exch store store
PaintRoot
```

2.4.5 Saving Keystrokes

If you often connect to the server directly, you can redefine commonly used commands to save typing.

Note: You should not use these shortened names in client programs since the definitions will not exist on other machines.

The following code added to your *user.ps* redefines several common commands to save keystrokes.

```
% Some aliases
/ps {pstack} def
/cds {countdictstack =} def
/fb framebuffer def
(keystroke savers ps, cds, fb defined\n) print
```

3. NeWS Extensions to PostScript

NeWS contains extensions to PostScript (as specified by Adobe in the *PostScript Language Reference Manual*), including new types and new operators. PostScript was initially designed for driving printers. As a part of NeWS, PostScript has to deal with multiple asynchronous clients, interaction, displays, keyboards, and locators. This chapter is an overview of NeWS's extensions to PostScript in the areas of lightweight processes, canvases, colors and cursors. For a detailed description of new PostScript types and operators, see Chapter 4, ''New PostScript Types'', and Chapter 5, ''New PostScript Operators''.

If all you want to do is write PostScript client programs, you will probably not be concerned with most of the primitives described here; you will instead want to use the packages that have been defined using these primitives to support windows, menus, and the like. These packages of PostScript code are described in Section 3 of this manual, ''Programming the IRIS Window Manager''. However, the primitives described here will be very useful if you are thinking of modifying the user interface of the window manager supplied with 4Sight, or writing your own window management routines.

3.1 The Lightweight Process Mechanism

The NeWS server maintains a set of simultaneously executing lightweight PostScript processes. Each process is an individual thread of control with its own graphics context, dictionary stack, execution stack, and operand stack. These processes all exist in the same address space; two processes can refer to the same object if they can both locate the object. Typically, each connection to the server obtains a separate thread of PostScript execution, with its own context. This thread can fork new threads and form a group of PostScript processes. Such groups of processes are represented by process objects.

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

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

Note: The current scheduling policy might change to a pre-emptive scheduling policy in the future. You should avoid writing PostScript code that relies on the behavior of a non-pre-emptive scheduling policy. PostScript code that accesses shared data structures should use monitors to protect those data structures.

3.2 Canvases and Shapes

A *canvas* is a surface upon which PostScript images are rendered. Each PostScript process has a canvas associated with it called the *current canvas*. Canvases exist in a hierarchy. At the root of a hierarchy is a device canvas. This device canvas is created as the result of the **createdevice** operator and represents the background, sometimes called the desktop. Additional canvas objects may be created with **newcanvas** calls. Canvases are accessed as dictionaries.

A canvas can be repositioned in the list of its siblings with **canvastotop** and **canvastobottom**. Its *x,y* offset relative to its parent may be set with **movecanvas**.

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

3.2.1 Visibility

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

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

3.2.2 Damage

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

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

Damage can be spread with **copyarea**, which copies a region on the canvas from one place to another. If part of the source is damaged, the corresponding destination area becomes damaged.

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

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

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

3.2.3 Event Consumption

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

3.2.4 Offscreen

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

3.2.5 Cursor

Every canvas has a *cursor* image that is displayed whenever the mouse cursor is over the canvas. This image is set with **setcanvascursor** and retrieved with **getcanvascursor**. A child canvas inherits its parent's cursor upon creation of the canvas. A cursor image is composed of a black *primary image* and a white *mask image*. These images are superimposed on their origins. These images are nothing more than characters in a font. The *hot spot* of a cursor (the pixel coordinate to which the mouse is pointing) is the origin of the primary image's character.

The current location of the cursor in the current coordinate system is returned by **currentcursorlocation**. Similarly **setcursorlocation** will move the cursor to a new location. The use of **setcursorlocation** is discouraged, since cursor jumps are usually disconcerting.

3.3 Colors

PostScript has a notion of color, which is implemented in the **sethsbcolor** and **setrgbcolor** primitives. NeWS extends this notion by adding a color type. Objects of type **color** can be created using either the HSB or the RGB color model. Color objects can also be compared with the **contrastswithcurrent** primitive.

4. New PostScript Types

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

4.1 New Objects in NeWS

The following objects are extensions to PostScript:

canvas

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

color

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

event

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

graphicsstate

Graphics state objects preserve entire PostScript graphics states in a permanent form. Their only use is to be saved and restored.

monitor

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

process

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

shape

Shape objects represent PostScript paths in a permanent form. Their only use is to be saved and restored; only the current path may be operated on.

4.2 Objects as Dictionaries

The internal state of some of these new types is accessible as if the object were a dictionary; fields in the object are accessed just like keys in a dictionary. For example, the first line of PostScript code below determines if **MyCanvas** is a colored canvas; the second sets the **Name** in **MyEvent** to **EnterEvent**.

```
MyCanvas /Color get
MyEvent /Name /EnterEvent put
```

Note: The use of new type objects as dictionaries has defined behavior only for the existing keys given for each type. You should not define new keys in these ''dictionaries''; the results are undefined and what happens may change in future implementations.

The following four sections describe the keys of interest in those types that are accessible as dictionaries. Those fields which cannot take direct input are labeled **read only**.

4.2.1 Canvases as Dictionaries

The accessible keys of a **canvas** dictionary are explained in this section.

Key	Type	Semantics
TopCanvas	*canvas*	The canvas at the top of the scene containing this canvas. **Read only.**
BottomCanvas	*canvas*	The canvas at the bottom of the scene containing this canvas. **Read only.**
CanvasAbove	*canvas/null*	The canvas immediately above this canvas, or null if no such canvas exists. **Read only.**
CanvasBelow	*canvas/null*	The canvas immediately below this canvas, or null if no such canvas exists. **Read only.**
TopChild	*canvas/null*	The top child of this canvas, or null if no such canvas exists. **Read only.**
Parent	*canvas/null*	The parent of this canvas, or null if the canvas has no parent. Null is returned, for example, for canvases that result from **createdevice**. Setting this field manipulates the window hierarchy.
Transparent	*boolean*	True if the canvas is transparent, false if it is opaque. An opaque canvas visually hides all canvases underneath it; a transparent canvas does not. A transparent canvas never has a retained image; instead it shares its parents retained image.

Table 4-1. Accessible Canvas Keys

Key	Type	Semantics
Mapped	*boolean*	True if the canvas is mapped, false if it is unmapped. When a canvas is mapped it becomes visible on the screen that its parent is on. When a nonretained window is mapped, the region that becomes visible is considered to be damaged.
Retained	*boolean*	True if the canvas is retained, false if it is not. NeWS keeps an offscreen copy of a retained canvas. If it is on a screen and overlapped by some other canvas, the hidden parts of the canvas will be saved. A retained canvas generally performs much better with most window management operations, like moving and popping canvases. But the retained image does consume storage. For color displays, the cost of retaining canvases is often prohibitive.
SaveBehind	*boolean*	A hint to the window system that when the canvas is made visible on the screen, any canvases below it won't be very active and the new canvas won't be up for very long. This is a performance hint only; it does not affect the semantics of any other operations. It is generally employed with pop-up menus to reduce the cost of damage repair when they are removed.
Color	*boolean*	True if and only if the current canvas can support more colors than just black and white, or greyscale. **Read only.**

Table 4-1. (continued) Accessible Canvas Keys

Key	Type	Semantics
EventsConsumed	*keyword*	The event consumption behavior of the canvas is determined by its **EventsConsumed** key, where *keyword* is either **/AllEvents**, **/MatchedEvents**, or **/NoEvents.** For definitions of these keywords, see Table 3-3.
Interests	*array*	The interest list for the canvas is returned as an *array* of events. The order of events in the array is the priority of the order of the interests, highest first. The **globalinterestlist** primitive returns an equivalent array of interests for the global interest list– the set of interests expressed with a null canvas. See Chapter 5, ''New PostScript Operators''.

Table 4-1. (continued) Accessible Canvas Keys

Keyword	Definition
/AllEvents	No events will be matched against canvases behind this one.
/MatchedEvents	Events that match an interest on this canvas will not be matched against any canvases behind this one.
/NoEvents	Events will be matched against interests on canvases behind this one, regardless of whether they match.

Table 4-2. EventsConsumed Keyword Definitions

4.2.2 GL Canvas Extensions

The following canvas fields were added to NeWS by Silicon Graphics, Inc. to include support for clients accessing the Graphics Library. These fields should be altered only by experienced programmers.

Key	Type	Semantics
GLCanvas	*boolean*	True if the canvas is a GL canvas. When **GLCanvas** is false, access to the other GL canvas fields is disabled.
GLCicnum	*number*	The number of the GL client's input channel.
GLCgfnum	*number*	The index of the GL/window context for the client that is using the canvas.
GLCpid	*number*	The process ID of the GL client using the canvas.
GLCfullsrcn	*boolean*	True if the canvas is in full screen mode.

Table 4-3. Accessible GL Canvas Keys

4.2.3 Events as Dictionaries

The currently accessible keys of an **event** dictionary are explained in this section.

Key	Type	Semantics
Action	*object*	An arbitrary PostScript object, often depending on the value of the **Name**. For keystrokes, the value of **Action** is **/DownTransition** or **/UpTransition**; for mouse motion, **Action** is null. In an interest, the **Action** may be a number, a keyword, or a string, in which case it is matched exactly against the **Action** of candidate events; or the **Action** may be an array or dictionary.
Canvas	*null/canvas*	When an event is submitted for distribution (by **sendevent** or **redistributeevent**), this key indicates a restriction on its distribution: the event only matches interests expressed with respect to the given canvas. Certain system events (such as **/Damaged** events) have a canvas specified. When an event is actually matched to an interest expressed with respect to a canvas (whether or not the event originally contained that canvas), this key is set to that canvas. In an interest, the **Canvas** specifies that only events that happen to that canvas will be matched. Null in an interest **Canvas** indicates an interest in all events not explicitly directed to a particular canvas.
ClientData	*object*	In either an interest or an event submitted for distribution, this field may hold additional information relating to the event. The information is carried along without modification by NeWS.

Table 4-4. Accessible Event Keys

Key	Type	Semantics
Exclusivity	*boolean*	**Exclusivity** is meaningful only for interests, although it may be set and read in any event (since any event may be used as an argument to **expressinterest**). If true, it indicates that an event that matches this interest in distribution should not be allowed to match any further interests.
Interest	*event*	This key is set in a real event as it is distributed; its value is the interest that the event matched in order to be delivered to its recipient. **Read only.**
IsInterest	*boolean*	This key indicates whether an event is currently on some interest list. **Read only.**
IsQueued	*boolean*	This key is true only when the event has been put in the input queue and has not yet been delivered. **Read Only.**
KeyState	*array*	When keyboard translation is on, this array is empty. When translation is off, this array indicates all the keys that were down at the time the event was distributed. (Normally, **liteUI.ps** is loaded by the server at initialization and it turns translation on. You can turn it off and do translation entirely in PostScript.) The array actually contains the **Name** values from events that had an **Action** of **/DownTransition**, and that did not have a subsequent event with the same **Name** and an **Action** of **/UpTransition.** In generating this array, the test is executed before a down-event, and after an up-event, so a down-up pair with no intervening events will not be reflected in the **KeyState** array. This key is meaningless in an interest. **Read only.**

Table 4-4. (continued) Accessible Event Keys

Key	Type	Semantics
Name	*object*	An arbitrary PostScript object, generally indicating the kind of event. For example, keystrokes will have numeric **Names** corresponding to the ASCII characters (or the keys) that were pressed. Many other events have keyword values, such as **/Damaged** or **/EnterEvent**. In an interest, the **Name** may be a number, a keyword, or a string, in which case it is matched exactly against the **Name** of candidate events; or it may be an array or dictionary.
Priority	*number*	**Priority** is meaningful only for interests, although it may be set and read in any event (since any event may be used as an argument to **expressinterest**). Real events are matched against the interests expressed on a **Canvas** in priority order, highest priority first; among interests with the same priority, the most recent is considered first. For these purposes, the global interest list (interests expressed with a null **Canvas**) is treated like the foremost canvas interest list. The default priority is 0; fractional and negative values are allowed, and there are very few circumstances where the priority need be changed at all.
Process	*null/process*	In the event queue, this key indicates the only process the event will be delivered to (if any — it must still match on all other criteria). In an interest list, it identifies the process that expressed this interest.
Serial	*number*	This key is a number that reflects the order in which events are taken off the event queue. For an interest, **Serial** equals the serial number of the last delivered event that matched this interest. **Read only.**

Table 4-4. (continued) Accessible Event Keys

Key	Type	Semantics
TimeStamp	*number*	This numeric value indicates the time an event occurred. (A time value is simply the number of minutes since the system started; it may contain a fractional component.) Events in the event queue are distributed in **TimeStamp** order, and no event is delivered before the time in its **TimeStamp** key. Thus, a timer event is simply any event handed to **sendevent** with a **TimeStamp** value in the future. This key is ignored in interests. The current nominal resolution of a time value is 2^{-16} seconds (about 0.9 ms) and the maximum interval is 65,536 minutes (about 45½ days).
XLocation	*number*	System events are labeled with the cursor location at the time they are generated; this value is used to determine which canvas(es) the event can be distributed to. It is available to recipients and is transformed to the current canvas' coordinate system. This key accesses the X-coordinate of the location. It is ignored in interests.
YLocation	*number*	This key accesses the Y-coordinate of the event location; see the explanation under **XLocation** above. It is ignored in interests.
AbsXLocation	*number*	This key accesses the X-coordinate of the event location in untransformed screen coordinates; it is provided to support *mex* compatibility. Its use is not recommended, and it may be removed in future releases. **Silicon Graphics extension.**
AbsYLocation	*number*	This key accesses the Y-coordinate of the event location in untransformed screen coordinates; it is provided to support *mex* compatibility. Its use is not recommended, and it may be removed in future releases. **Silicon Graphics extension.**

Table 4-4. (continued) Accessible Event Keys

4.2.4 Processes as Dictionaries

The keys that may be accessed in a **process** dictionary are explained in this section. All the keys in this section are read only; attempts to change their values in a process raise **unregistered** errors.

Key	Type	Semantics
DictionaryStack	*array*	The current dictionary stack of the process is returned as an array. The earliest dictionary is array element 0. **Read only.**
ErrorCode	*keyword*	The current errorcode of the process is returned as a keyword. The set of possible results is listed in Table 3-10. **Read only.**
ErrorDetailLevel	*process*	Controls the amount of detail that is included in an error report. Setting **errorDetailLevel** to 0 (the default) gives a minimum of error reporting. Setting it to 1 gives a more detailed message, 2 dumps the contents of the dictionary, execution,and operand stacks. Setting the detail level is done as follows: `currentprocess /ErrorDetailLevel 1 put.`
Execee	*object*	The object currently being evaluated (i.e., the top of the process' execstack) is returned. **Read only.**
Executionstack	*array*	The full current execution stack of the process is returned as an array, containing pairs of executable arrays and indices. The latest executable array is element 0 of the array, the ‘‘program counter'' within it is element 1. **Read only.**
Interests	*array*	The current interest list of the process is returned as an array. The first element of the array is the event which is the most recently expressed interest in this process. **Read only.**

Table 4-5. Accessible Process Keys

Key	Type	Semantics
OperandStack	*array*	The full current operand stack of the process is returned as an array. The earliest object on the stack is element 0. **Read only.**
StandardErrorNames	*array*	An array of the names of the standard errors. It is used by **errored** and by the debugger. It is available for other programs' use.
State	*keyword*	The current execution state of the process is returned as a keyword. The set of possible results is listed in Table 4-6. **Read only.**

Table 4-5. (continued) Accessible Process Keys

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

Table 4-6. State Keywords

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

Table 4-7. ErrorCode Keywords

4.3 Opaque Objects

The following object types are not accessible as dictionaries:

- **color**
- **graphicsstate**
- **shape**
- **monitor**

Color objects are used only to store color values. Graphics state objects are used only to save the graphics state of a process for future use by that (or another) process. Shape objects are used only to save the current path of a process for future use by that (or another) process. Monitor objects are used only for mutual exclusion.

4.4 Object Cleanup

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

4.4.1 Server Function

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

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

The following code does this.

```
100 dict begin initmatrix newprocessgroup
```

Finally, the client code is executed by executing:

connection file `cvx exec`

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

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

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

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

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

The client will have altered the **/NestedServer** loop to add the additional process **/AbortProc**. This code will be executed in the initial client process when the socket is closed. Some resources, such as cached fonts, are considered system resources and are not under client control.

4.4.2 Object Management

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

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

4.4.3 Error Handling

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

Client side errors are dependent on their environment. C clients have to do most of their own error-handling. The C client can catch signals and can install a cleanup proc for their server connection using the **AbortProc** and **NestedServer** (discussed above in Section 4.4.1).

4.4.4 Connection Management

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

4.4.5 Process Management

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

Killing An Application

An application should kill its process groups and any other process groups it has created upon its own termination. The NeWS server does this when your client socket closes.

The *psh* client creates a new process group so that the closing of the client socket does not automatically kill the client. Otherwise, all *psh* clients would always immediately die as soon as their window was created.

Garbage Collection

Garbage collection is the process of removing objects from virtual memory when they are no longer referenced. The problem is quite acute in the current generation of printers that understand the PostScript and any virtual memory system must cope with it. Killing the client process group will "dereference" the client's userdict, which will recursively garbage collect all the client's local references. When the main process dies, it kills its process group. Only if other process groups have been created is there a need to explicitly kill the forked processes made by that group.

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

4.4.6 De-Referencing Composite Objects

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

4.5 NeWS Security

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

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

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

5. New PostScript Operators

This chapter contains detailed information about NeWS operator extensions to the PostScript language.

The chapter is divided into two sections. The first section contains a summary of the operators grouped according to function. The second section contains detailed descriptions of all the operators, and is organized alphabetically.

5.1 New Operator Summary

The following sections summarize the NeWS operator extensions to PostScript. They are grouped according to function.

5.1.1 Arithmetic and Math Operators

num1 **arccos** num2	arc cosine (in degrees) of *num1*
num1 **arcsin** num2	arc sine (in degrees) of *num1*
num1 **arctan** num2	arc tangent (in degrees) of *num1*
a b **max** c	maximum of *a* and *b*
a b **min** c	minimum of *a* and *b*
– **random** num	random number in range [0,1]

5.1.2 Canvas Operators

w h bps mat proc **buildimage** canvas	construct *canvas*
canvas **canvastobottom** –	move *canvas* to bottom of sibling list
canvas **canvastotop** –	move *canvas* to top of sibling list
– **clipcanvas** –	clip current canvas
– **clipcanvaspath** –	set current path to canvas clipping
string **createdevice** canvas	create *canvas* from device '*string*'
canvas1 **createoverlay** canvas2	create overlay canvas
– **currentcanvas** canvas	get current value of *canvas*
– **eoclipcanvas** –	clipping with even/odd winding
– **eoextenddamage** –	add current path to damage, even/odd
canvas **eoreshapecanvas** –	reshape with even/odd winding
file *or* string **eowritecanvas** –	saves canvas image in a file
file *or* string **eowritescreen** –	saves screen dump in a file
– **extenddamage** –	add current path to damage
canvas **getcanvascursor** font ch1 ch2	get cursor id's for *canvas*
canvas **getcanvaslocation** x y	loc. of *canvas* relative to current canvas
canvas **imagecanvas** –	render *canvas* on current canvas
boolean canvas **imagemaskcanvas** –	render *canvas* with mask
canvas x y **insertcanvasabove** –	insert current canvas above *canvas*
canvas x y **insertcanvasbelow** –	insert current canvas below *canvas*
x y **movecanvas** –	move current canvas relative to parent
pcanvas **newcanvas** ncanvas	create canvas whose parent is *pcanvas*
string **readcanvas** canvas	read file '*string*' into canvas
canvas **reshapecanvas** –	set shape of *canvas* to current path
canvas **setcanvas** –	set current canvas to *canvas*
canvas font ch1 ch2 **setcanvascursor** –	set cursor id's for *canvas*
file *or* string **writecanvas** –	saves canvas image in a file
file *or* string **writescreen** –	saves screen dump in a file

5.1.3 Color Operators

color **contrastswithcurrent** boolean	true if *color* is not current color
– **currentcolor** color	return current color
h s b **hsbcolor** color	return hsb color object
r g b **rgbcolor** color	return rgb color object
color **setcolor** –	set current color to *color*

5.1.4 Communication Operators

– **setfileinputtoken** –	define compressed tokens
n **tagprint** –	print encoded tag value
o **typedprint** –	print encoded object to output stream

5.1.5 Dictionary Operators

dict key **undef** –	remove definition of *key* from *dict*

5.1.6 Event Operators

– **awaitevent** event	block process until *event* occurs
num **blockinputqueue** –	block input queue *num* minutes
– **countinputqueue** num	return number of events on input queue
– **createevent** event	create event object
event **expressinterest** –	queue events of type *event*
– **globalinterestlist** array	get *array* of null canvas events
– **lasteventtime** num	return **TimeStamp** of latest event
event **recallevent** –	remove *event* from input queue
event **redistributeevent** –	return received *event*
event **revokeinterest** –	ignore events of type *event*
event **sendevent** –	send *event* to event queue
– **unblockinputqueue** –	unblock input queue if blocked

5.1.7 File Operators

listenfile **acceptconnection** file	listen for connection to NeWS
– **dumpsys** –	dump system state to standard output
string1 string2 **file** file	open file '*string1*' with access *string2*

5.1.8 Font Operators

– **enumeratefontdicts** names	put existing font names on stack

5.1.9 Graphics State Operators

– **currentcursorlocation** x y	return cursor position
– **currentlinequality** n	return current line quality value
– **currentprintermatch** boolean	return value of **printermatch** flag
– **currentrasteropcode** num	return rasterop function
– **currentstate** state	return current graphics state
n **setlinequality** –	set current line quality to *n*
boolean **setprintermatch** –	set **printermatch** flag
num **setrasteropcode** –	set current rasterop function
graphicsstate **setstate** –	set currnet graphics state

5.1.10 Image Operators

dx dy **copyarea** –	copy area relative to its current position
dx dy **eocopyarea** –	as above, with even/odd winding

5.1.11 Monitor Operators

– **createmonitor** monitor	create new monitor object
monitor procedure **monitor** –	execute *procedure* with *monitor* locked
monitor **monitorlocked** boolean	true if *monitor* is locked

5.1.12 Path Operators

– **currentpath** shape	return *shape* describing current path
– **damagepath** –	set current path to damage path
– **emptypath** boolean	true if current path is empty
– **eocurrentpath** shape	return *shape* using even/odd rule
array **pathforallvec** –	create path using *array* of procs
x y **pointinpath** boolean	true if point [*x*,*y*] is in current path
path **setpath** –	set current path to *path*

5.1.13 Process Operators

proc **activate** process	create new process
– **breakpoint** –	suspend current process
process **continueprocess** –	restart suspended *process*
– **currentautobind** boolean	true if autobinding is enabled
– **currentprocess** process	return current process
procedure **fork** process	run *procedure* in copy of current process
– **geteventlogger** process	return current event logger
process **killprocess** –	kill *process*
process **killprocessgroup** –	kill processes in same group as *process*
– **newprocessgroup** –	create new process group
– **pause** –	halt current process until rest are done
boolean **setautobind** –	enable/disable autobinding
process **seteventlogger** –	make *process* the event logger
process **suspendprocess** process	suspend *process*
process **waitprocess** value	return *value* when *process* completes

5.1.14 Server Control Operators

– **^C** –	abort NeWS server
– **startkeyboardandmouse** –	initialize input devices
any **errored** bool	establish context for catching errors

5.1.15 Server State Operators

– **currenttime** num	return time in minutes
x y **setcursorlocation** –	move cursor to (*x,y*) in current canvas

5.1.16 IRIX Operators

string1 **canonicalizehostname** string2	convert host nickname to official name
string **forkunix** –	execute *string* as shell command
string1 **getenv** string2	return shell environment variable
– **getkeyboardtranslation** num	return keyboard translation mode
file **getsocketlocaladdress** string	return *string* describing address of *file*
file **getsocketpeername** string	return name of *file*'s host
– **keyboardtype** num	return value representing keyboard type
– **localhostname** string	return official name of server's host
string1 string2 **putenv** –	set shell environment variable
num **setkeyboardtranslation** –	set keyboard translation mode

5.2 New Operator Details

– **^C** –

Abort the NeWS server. (The command consists of two characters – a '^' followed by a 'C' – not a control-C.)

listenfile **acceptconnection** file

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

proc **activate** process

Creates a new process that executes *proc* in an environment that is an exact copy of the original process's environment. *process* is a handle by which the newly-created process can be manipulated.

num1 **arccos** num2

Computes the arc cosine (in degrees) of *num1*.

num1 **arcsin** num2

Computes the arc sine (in degrees) of *num1*.

num1 **arctan** num2

Computes the arc tangent (in degrees) of *num1*. You should probably use **atan** instead.

– **awaitevent** event

Blocks the PostScript process until an event in which it has expressed interest happens, and returns an object of type *event* describing it.

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

num **blockinputqueue** –

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

Events used as arguments to **sendevent** are inserted in the event queue, and hence are subject to inhibition; events passed to **redistributeevent** are not re-queued, and so are not inhibited. See Section 7.6, ''Input Synchronization'', for more information.

See also: **sendevent, unblockinputqueue.**

– **breakpoint** –

Suspends the current process.

width height bitspersample matrix proc **buildimage** canvas

Constructs a canvas object from the *width*, *height*, *bitspersample*, and *proc* parameters in the same way as **image** interprets its parameters. A notable difference between the Sun and Adobe implementations is that if *bitspersample* is 24 then the image is interpreted as color. The inverse of the matrix is used to define the default coordinate system of the canvas.

The parameters represent a sampled image that is a rectangular array of *width* by *height* sample values, each of which consists of *bitspersample* bits of data (1,8,24). The data is received as a sequence of characters (i.e., 8-bit integers in the range 0 to 255). If *bitspersample* is less than 8, the sample bits are packed left to right within a character (from the right-order bit to the low-order bit). Each row is padded out to a character boundary.

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

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

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

string1 **canonicalizehostname** string2

Given a host nickname, **canonicalizehostname** returns the official network name of the host.

canvas **canvastobottom** –

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

canvas **canvastotop** –

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

– **clipcanvas** –

The **clipcanvas** operator is similar to **clip** except that it sets a clipping boundary that is an attribute of the current *canvas*, not the current graphics state. This clipping boundary is not affected by **initgraphics, initclip, gsave, grestore**, or any of the other graphics state modifiers. Graphics operations are clipped to the intersection of the canvas clip, the graphics state clip, and the shape of the canvas.

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

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

See also: **damagepath, clipcanvaspath.**

– **clipcanvaspath** –

Sets the current path to the canvas clipping for the current canvas as set by **clipcanvas.**

process **continueprocess** –

Restarts a suspended process.

See also: **suspendprocess, breakpoint.**

color **contrastswithcurrent** boolean

Returns true if the color argument is different than the current color. The test takes into account the characteristics of the current device. The standard boolean operators, like **eq** can be used to compare colors without accounting for the current device.

dx dy **copyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. For example, you could use this primitive to scroll a text window. The nonzero winding number rule is used to define the inside and outside of the path.

– **countinputqueue** num

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

string **createdevice** canvas

Creates a new canvas from a frame buffer device named by *string*. The values for *string* on an IRIS is `gl8` for 8-bit color, or `gl24` for 24-bit RGB color.

– **createevent** event

Synthesizes an object of type *event*, which will have null values for all its fields.

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

– **createmonitor** monitor

Creates a new monitor object.

See also: **monitorlocked, monitor.**

canvas1 **createoverlay** canvas2

Given *canvas1*, **createoverlay** creates a *canvas2* that overlays the original. An overlay is like a sheet of cellophane that lays over a canvas. Anything that is drawn in an overlay will float over the underlying canvas. Objects drawn in the overlay will not affect the underlying image, and objects drawn in the underlying image will not affect the overlay. Because of the way that overlays are implemented on some displays, there will be performance problems if too many things are written into the overlay. They are intended to be used for animated objects like rubber band lines and bounding boxes.

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

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

– **currentautobind** boolean

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

See also: **setautobind**.

– **currentcanvas** canvas

Returns the current value of the canvas parameter in the graphics state.

– **currentcolor** color

Returns the current color as set by **setcolor**, **setrgbcolor**, or **sethsbcolor**.

– **currentcursorlocation** x y

Returns the current position of the cursor (mouse) in user coordinates.

– **currentlinequality** n

Returns a number between 0 and 1 that represents the current line quality.

See also: **setlinequality.**

– **currentpath** shape

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

– **currentprintermatch** boolean

Returns the current value of the **printermatch** flag in the graphics state.

See also: **setprintermatch.**

– **currentprocess** process

Returns an object that represents the current process.

– **currentrasteropcode** num

Returns a number that represents the current rasterop combination function.

See also: **setrasteropcode.**

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

– **currentstate** state

Returns a **graphicsstate** object that is a snapshot of the current graphics state. See also: **setstate.**

– **currenttime** num

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

– **damagepath** –

Sets the current path to be the damage path from the current canvas. The damage path will be cleared. The damage path represents those parts of the canvas that were damaged by some manipulation of the scene on the display, and that could not be repainted from stored bitmaps.

Processes can arrange to be notified of damage through the input mechanism. Whenever damage occurs to a canvas, a **Damaged** event will be generated.

See also: **clipcanvas.**

– **dumpsys** –

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

– **emptypath** boolean

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

– **enumeratefontdicts** names

Scans through all the font dictionaries that NeWS knows about and pushes the name (as a keyword object) of each font family file that it can find.

Note: PostScript code should use **FontDirectory** in preference to **enumeratefontdicts.**

– **eoclipcanvas** –

This is the same as **clipcanvas** except that it uses the even/odd winding number rule rather than the nonzero rule.

See also: **clipcanvas.**

dx dy **eocopyarea** –

Copies the area enclosed by the current path to a position offset by *dx,dy* from its current position. For example, you could use this primitive to scroll a text window. The even/odd winding number rule is used to define the inside and outside of the path.

See also: **copyarea.**

– **eocurrentpath** shape

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

See also: **currentpath.**

– **eoextenddamage** –

Adds the current path to the damage shape for the current canvas. A **/Damaged** event is sent to those processes that have expressed interest. Uses the even-odd winding rule.

canvas **eoreshapecanvas** –

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

See also: **reshapecanvas.**

file *or* string **eowritecanvas** –

Either opens *string* as a file for writing, or if the argument is a *file*, simply writes to it. Creates a rasterfile which contains an image of the region outlined by the current path in the current canvas. If the current path is empty, the whole canvas is written. **eowritecanvas** is used to save an image in a file. It follows an even-odd winding rule rather than a non-zero winding rule.

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

file *or* string **eowritescreen** –

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

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

any **errored** bool

errored acts just like the **stopped** PostScript primitive, but for errors. Since this is generally what **stopped** has been used for, **errored** is recommended as a direct replacement.

event **expressinterest** –

Input events matching *event* will be queued for reception by **awaitevent.**

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

– **extenddamage** –

Adds the current path to the damage shape for the current canvas. A **/Damaged** event is sent to those processes that have expressed interest. Uses the non-zero winding rule.

string1 string2 **file** file

Identical to the Adobe PostScript implementation, with one exception: if the file identified by *string1* cannot be found, and it is not an absolute pathname, the server will attempt to open the file *string1* in */usr/NeWS/lib.*

procedure **fork** process

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

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

string **forkunix** –

Forks an IRIX process to execute *string* as a shell command line. Standard input and output are directed to */dev/null.*

canvas **getcanvascursor** font char1 char2

Gets the cursor identifiers for *canvas. font* is the font where the cursor image characters (primary and mask) are stored. The *char1* is used as the index to locate the primary image and *char2* is used to locate the mask image.

See also: **setcanvascursor.**

canvas **getcanvaslocation** x y

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

See also: **movecanvas.**

string1 **getenv** string2

Returns the value of the variable *string1* in the environment of the server process, as modified by any **putenv** operations. This operator fails with an undefined error if *string1* is not present in the environment. One can guard against this case by using the **stopped** operator to recover from the error, as in the example below.

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

See also: **putenv**.

– **geteventlogger** process

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

See also: **seteventlogger**.

– **getkeyboardtranslation** num

Returns a small integer that indicates the mode of translation being performed on keyboard input by the underlying operating system. Normally, the return value will be 2 if the operating system is translating events into ASCII characters and escape sequences, or 3 if it is not. In either case, input events are generated with a name field containing an integer equal to 0x6F00 + the byte emitted by the keyboard.

See also: **keyboardtype**, **setkeyboardtranslation**.

file **getsocketlocaladdress** string

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

file **getsocketpeername** string

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

– **globalinterestlist** array

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

h s b **hsbcolor** color

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

canvas **imagecanvas** –

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

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

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

boolean canvas **imagemaskcanvas** –

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

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

canvas x y **insertcanvasabove** –

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

canvas x y **insertcanvasbelow** –

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

– **keyboardtype** number

Returns a small integer indicating the kind of keyboard attached to the NeWS server. The return values are documented under *keyboard*(7).

See also: **getkeyboardtranslation, setkeyboardtranslation.**

process **killprocess** –

Kills *process*.

process **killprocessgroup** –

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

See also: **newprocessgroup.**

– **lasteventtime** num

Returns the **TimeStamp** of the last event delivered by the input system.

– **localhostname** string

Returns the official network hostname of the host on which the server is running.

a b **max** c

Compares *a* and *b* and leaves the maximum of the two on the stack. Works on any data type for which **gt** is defined.

a b **min** c

Compares *a* and *b* and leaves the minimum of the two on the stack. Works on any data type for which **gt** is defined.

monitor procedure **monitor** –

Executes *procedure* with *monitor* locked (**entered**). At most one process may have a monitor locked at any one time. If a process attempts to lock a locked monitor it will block until the monitor is unlocked. If an error occurs during the execution of *procedure* and the execution stack is unwound beyond the *monitor*, then the *monitor* object will be unlocked.

See also: **createmonitor, monitorlocked.**

monitor **monitorlocked** boolean

Returns true if the *monitor* is currently locked; false otherwise.

See also: **createmonitor, monitor.**

x y **movecanvas** –

Moves the current canvas to (*x*,*y*) relative to its parent. (*x*,*y*) is a delta vector interpreted according to the current transformation. This motion is relative to the lower left-hand corner of the two canvases – (0,0) interpreted with reference to the initial matrix for each canvas. The *mapped* attribute of the canvas does not change.

See also: **getcanvaslocation.**

pcanvas **newcanvas** ncanvas

Creates a new empty canvas, *ncanvas*, whose parent is *pcanvas.*

Note: These defaults are the result of historical precedent. To ensure the portability of your programs to future releases of the system, you should always explicitly set the **Transparent** property of all new canvases.

It defaults to being opaque if its parent is the framebuffer; transparent otherwise. It defaults to being retained if it is opaque and the number of bits per pixel of the framebuffer is less that the retain threshold. If your program relies on having a canvas be retained you should explicitly set it to be retained.

See also: **reshapecanvas.**

– **newprocessgroup** –

Creates a new process group with the current process as its only member. When a process forks the child will be in the same process group as its parent.

array **pathforallvec** –

The single argument to **pathforallvec** is an array of procedures. The **pathforallvec** operator then enumerates the current path in order, executing one of the procedures out of the array for each of the elements in the path. The type of the path element determines which array element will be executed. **moveto, lineto, curveto,** and **closepath,** respectively, are array elements 0, 1, 2, and 3. If the array is too short, **pathforallvec** will try to reduce elements of one type to another. Array element 5 is used to handle conic control points. The standard PostScript operator **pathforall** is exactly equivalent to '`4 array astore pathforallvec`.' For further information, consult the PostScript Language Reference Manual description of the **pathforall** operator. Users are cautioned against using this primitive if at all possible, and using **pathforall** instead.

– **pause** –

Suspends the current process until all other eligible processes have had a chance to execute.

x y **pointinpath** boolean

Returns true if the point (*x*,*y*) is inside the current path.

string1 string2 **putenv** –

Defines the shell environment variable *string1* to have the value *string2*. The environment variables inherited by the server as modified by **putenv** calls are inherited by IRIX processes created as children of the server with **forkunix**.

See also: **getenv**.

– **random** num

Returns a random number in the range [0,1].

string **readcanvas** canvas

String must be the name of a file in the server's file name space. The file must be in Sun raster file format or IRIS RGB image file format. If both these conditions are true, a retained, opaque canvas will be created. The canvas will have the depth specified in the raster file, will not have a parent, and will not be mapped. The canvas *cannot* be mapped; an **invalidaccess** error will result if you try to map the canvas. The canvas is useful only as a source for **imagecanvas**. If the file can't be found, an **undefinedfilename** error is generated. If the file can't be interpreted as a raster file, an **invalidaccess** error is generated.

event **recallevent** –

The event passed as an argument is removed from the event queue. The most common use for this primitive is to turn off a timer-event that has been sent but not yet delivered.

See also: **sendevent**.

event **redistributeevent** –

Return an event that has been received by the calling process to the distribution mechanism, which will continue as though the event had not matched the interest which gave it to this process.

See also: **expressinterest**.

canvas **reshapecanvas** –

Sets the shape of *canvas* to be the current path, and it sets the canvas' default transformation matrix from the current transformation matrix. This also establishes its position relative to the current canvas. If *canvas* is the same as the current canvas, then an implicit **initmatrix** will be done. The entire contents of the canvas is considered to be damaged.

The **initclip** operation will set the path to the shape defined by the shape of the current canvas.

Think of the current transformation matrix as laying down a grid over the current path. This grid has its origin somewhere relative to the path and it has some scale, rotation, and skew associated with it. When **reshapecanvas** sets the default transformation matrix for the canvas, it sets it so that this same grid is laid over the canvas as is laid over the current path, with the origin in the same relative location.

event **revokeinterest** –

No more input events matching *event* will be distributed to this PostScript process.

See also: **expressinterest**.

r g b **rgbcolor** color

Takes three numbers between 0 and 1 representing the red, green, and blue components of a color and returns a *color* object that represents that color.

event **sendevent** –

Submit an event to the input distribution mechanism (i.e., sort it into the event queue according to its **TimeStamp**).

See also: **awaitevent, createevent, recallevent, redistributeevent, expressinterest.**

boolean **setautobind** –

Enables or disables autobinding for the current process. By default it is on. See Appendix B, ''Autobind'', for more information on autobinding.

See also: **currentautobind.**

canvas **setcanvas** –

Sets the current canvas to be *canvas*. Implicitly executes **newpath initmatrix.**

canvas font char1 char2 **setcanvascursor** –

Sets the cursor identifiers for *canvas*. *font* is the font where the cursor image characters (primary and mask) are stored. *char1* is used as the index to locate the primary image and *char2* is used to locate the mask image.

See also: **getcanvascursor.**

color **setcolor** –

Sets the current color to be *color*. **rgbcolor setcolor** is the same as **setrgbcolor**, and **hsbcolor setcolor** is the same as **sethsbcolor.**

x y **setcursorlocation** –

Moves the cursor so its hot spot is at (*x*, *y*) in the current canvas' coordinate space.

process **seteventlogger** –

process (which must have expressed some interest – it doesn't matter what) is made the event-logger. Thereafter, a copy of every event that enters the distribution mechanism will be given to this process prior to (and without affecting) the rest of the distribution mechanism. This facility is offered as a PostScript debugging aid.

See also: **geteventlogger**.

– **setfileinputtoken** –

Used to define compressed tokens for communication efficiency.

number **setkeyboardtranslation** –

Instructs the underlying operating system to switch to the keyboard translation mode indicated by *number*. Normal values are TR_EVENT (2), meaning do translation, or TR_UNTRANS_EVENT (3), meaning provide unencoded up/down values for keyboard events. The default is TR_EVENT.

See also: **keyboardtype, getkeyboardtranslation.**

n **setlinequality** –

Sets the current desired line quality to *n*, which must be a number from 0 to 1. Line quality controls the quality of lines rendered by the **stroke** primitive. Increasing values of line quality increase the quality of the rendered line, and decrease performance. A value of 0 renders lines as fast as possible with the least attention paid to quality (the line thickness is ignored, lines are always a single pixel wide). A value of 1 renders lines with the highest possible quality: they will be the correct width, and all endcaps and joins will be correct. Intermediate values may give you different quality/performance tradeoffs. The default value for line quality is 0.

See also: **currentlinequality**.

path **setpath** –

Sets the current path from the shape object *path.*

boolean **setprintermatch** –

Sets the current value of the **printermatch** flag in the graphics state to *boolean.* When printer matching is enabled text output to the display will be forced to match exactly text output to a printer. The metrics used by the printer will be imposed on the display fonts. This will usually cause displayed text to look bunched up and generally reduce readability. With printer matching disabled readability will be maximized, but the character metrics for the display will not correspond to the printer.

See also: **currentprintermatch.**

num **setrasteropcode** –

Sets the current rasterop combination function, which will be used in subsequent graphics operations. The values that **setrasteropcode** takes are the same as the RasterOp function codes used by the Sun Pixrect library, though they must be calculated. Useful values are *PIX_NOT(PIX_ExT)* = **5**, *PIX_SRC^PIX_ExT* = **6**, *PIX_SRC|PIX_ExT* = **14**, etc.

Note: The RasterOp combination function exists only to support emulation of existing window systems. The definition of rasterop is display-specific. Currently, the **image** and **copyarea** primitives do not use the rasteropcode.

See also: **currentrasteropcode.**

graphicsstate **setstate** –

Sets the current graphics state from *graphicsstate.*

See also: **currentstate.**

– **startkeyboardandmouse** –

Initiate server processing of keyboard and mouse input. This is called once from early initialization code in *init.ps*, and should not be called again.

process **suspendprocess** process

Suspends the given process.

See also: **breakpoint, continueprocess.**

n **tagprint** –

Prints the integer *n*. This integer is encoded as a tag on the current output stream. Tags are used to identify packets sent from the PostScript server to client programs. See Chapter 8, "Mixing PostScript and C", for information on how the CPS input mechanism uses tags.

o **typedprint** –

Print the object *o* in an encoded form on the current output stream. These objects can then be read by client programs using the facilities of CPS. The format in which objects are encoded is described in Appendix B, "Byte Stream Format".

– **unblockinputqueue** –

An input queue lock set by **blockinputqueue** is released. If this reduces the count of locks to 0, distribution of events from the input queue is resumed. If the count was already 0, a rangecheck error is raised.

See also: **blockinputqueue.**

dictionary key **undef** –

Removes the definition (if any) of *key* from the *dictionary*.

process **waitprocess** value

Waits until *process* completes, and returns the value that was on the top of its stack at the time that it exited.

See also: **fork.**

file *or* string **writecanvas** –

Either opens *string* as a file for writing, or if the argument is a *file*, simply writes to it. Creates a rasterfile which contains an image of the region outlined by the current path in the current canvas. If the current path is empty, the whole canvas is written. **writecanvas** is used to save an image in a file.

See also: **writescreen, eowritecanvas, eowritescreen**

file *or* string **writescreen** –

Either opens *string* as a file for writing, or if the argument is a *file*, simply writes to it. Creates a rasterfile which contains an image of the entire screen. **writescreen** writes pixels from the screen and it will include pixels from canvases that overlap the current canvas. If the current path is empty, the whole canvas is written. **writecanvas** could be used to perform a screen dump as follows:

```
framebuffer setcanvas (/tmp/snap) writescreen
```

See also: **writecanvas, eowritecanvas, eowritescreen**

5.3 Non-Primitive PostScript Operators

Many useful operators can be defined using the primitive operators from the *PostScript Language Reference Manual* and the previous section. This section gives a partial listing of non-primitive operators that are defined in the PostScript files found in */usr/NeWS/lib/NeWS*. See Section 3 of this manual, ''Programming the IRIS Window Manager'' for more information on these files.

5.3.1 Miscellaneous Utilities

case and **append** are operations that nearly all PostScript programs need to perform. **sprintf/printf/fprintf** are near equivalents to their IRIX counterparts. **arrayinsert, arraydelete,** and **arrayop** are useful operations on arrays. **dictbegin/dictend** save you from counting the size of dictionaries. **modifyproc** brackets a procedure. **sleep** allows a PostScript procedure to sleep for an arbitrary period of time; **getvalue** and **setvalue** are useful for checking the status of an item.

obj1 obj2 **append** obj3

Concatenates arrays, strings, and dictionaries. In case of duplicate dictionary keys, the keys in the second dictionary overwrite the first's.

array index **arraydelete** –

Deletes the value in array at position *index*. If *index* is beyond the end of the array, the last item in the array is deleted. For example:

```
[/a /b 0 /x /y] 2 arraydelete ⇒ [/a /b /x /y]
```

array index value **arrayinsert** newarray

Creates a new array one larger than the initial array inserting *value* at position *index*. If *index* is beyond the end of the array, *value* is appended to the end of the array. For example:

```
[/a /b /x /y] 2 0 arrayinsert ⇒ [/a /b 0 /x /y]
```

A B proc **arrayop** C

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

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

value {key proc key key proc...} **case** –

Compares *value* against several keys, performing the associated procedure if a match is found. The key **/Default** matches all values. The following converts a number to a (whimsical) string:

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

– **dictbegin** –

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

– **dictend** dict

Returns the dictionary created by a previous **dictbegin**; together, they shrink-wrap a dictionary around your **defs**. For example:

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

file formatstring argarray **fprintf** –

Prints to *file*. For example:

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

will print the time the NeWS server has been running on your console.

See also: **console**.

– **getvalue** value

Returns the **ItemValue** of an item. The type of value depends upon the nature of the item.

formatstring argarray **printf** –

Prints on standard out, like **print**.

See also: **dbgprintf**.

value **setvalue** –

Sets the **ItemValue** of an item. The type of *value* depends upon the nature of the item.

interval **sleep** –

Sends itself an event **TimeStamped** *interval* in the future and returns when that event is delivered. *interval* is in minutes, with sixteen bits of decimal. The usable resolution is about 10 ms.

formatstring **sprintf** string

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

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

puts the string

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

on the stack.

proc head tail **modifyproc** {head proc tail}

Adds a *head* and *tail* modification to a procedure. Mainly used to override the behavior of a procedure. For example:

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

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

Note: Any of the procedures can be keywords.

5.3.2 User Interaction and Event Management

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

eventname eventproc action canvas **eventmgrinterest** interest

Makes an interest suitable for use by **forkeventmgr**. For example:

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

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

interests **forkeventmgr** process

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

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

x0 y0 procedure **getanimated** process

Forks a process that does animation while tracking the mouse, returning the process object *process* to the parent process. Each time the mouse is moved, the process executes **erasepage x0 y0 moveto,** pushes the current mouse coordinates *x* and *y* onto its stack, and calls *procedure*. The variables *x0*, *y0*, *x*, and *y* are available to *procedure*. After *procedure* returns, the process executes the **stroke** operator. Thus, your *procedure* can use *x0*, *y0*, *x*, and *y* to build a path that will be drawn each time the mouse is moved — drawing a line to the current cursor location, for example.

The process calling your *procedure* exits when the user clicks the mouse; it leaves the final mouse coordinates in an array [**x y**] on top of its stack, so that they are available to the parent process via the **waitprocess** operator. Since **erasepage** is executed each time the mouse is moved, the current canvas should be an overlay canvas when calling **getanimated.** Since **erasepage** is executed each time the mouse is moved, the current canvas should be an overlay canvas when calling **getanimated**. **getanimated** is used to implement most rubber-banding operations.

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

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

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

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

See also: **createoverlay, waitprocess.**

– **getclick** x0 y0

Uses **getanimated** to let the user indicate a point on the screen; it returns the location of a mouseclick to the stack.

x0 y0 **getrect** process

Uses **getanimated** to let the user "rubber-band" a rectangle with a fixed origin *x0, y0*. Returns a process with which you can retrieve the coordinates of the upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x1 y1] on the stack.

– **getwholerect** process

Uses **getclick** and **getrect** to let the user indicate both the origin and a corner of a rectangle. Returns a process with which you can retrieve the coordinates of the origin and upper right-hand corner of the rectangle. Use **waitprocess** to put these coordinates [x0 y0 x1 y1] on the stack.

/primary /mask canvas **setstandardcursor** –

Sets *canvas*'s cursor to the cursor composed of the *primary* and *mask* keywords. *primary* and *mask* must be cursors in cursorfont, the font of standard system cursors loaded by *cursor.ps*. For example:

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

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

Table 5-1 lists the cursors (and their masks) in cursorfont. See also: **setcanvascursor**.

Primary Image	Mask Image	Description	When/Where Used
ptr	**ptr_m**	arrow pointing to upper left	default
beye	**beye_m**	bullseye	window frame
rtarr	**rtarr_m**	→ arrow	menus
xhair	**xhair_m**	crosshairs (+ shape)	
xcurs	**xcurs_m**	× shape	icons
hourg	**hourg_m**	hourglass shape	start-up/canvas busy
nouse	**nouse_m**	no cursor	

Table 5-1. Standard NeWS Cursors

5.3.3 Rectangle Utilities

rect, rectpath, rect2points, rectsoverlap, and **insetrect** are useful utilities for managing rectangular coordinates and paths; other graphics procedures are in the next section, "Graphics Utilities".

delta x y w h **insetrect** x′ y′ w′ h′

Creates a new rectangle inset by *delta*.

x y x′ y′ **points2rect** x y width height

Converts a rectangle specified by its origin and top right corner to one specified by an origin and size.

width height **rect** –

Adds a rectangle to the current path at the current pen location.

x y width height **rect2points** x y x′ y′

Converts a rectangle specified by its origin and size to a pair of points specifying the origin and top right corner of the rectangle.

x y width height **rectpath** –

Adds a rectangle to the current path with *x,y* as the origin.

x y w h x′ y′ w′ h′ **rectsoverlap** boolean

Returns true if the two rectangles overlap.

5.3.4 Graphics Utilities

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

string **cshow** –

Shows *string* centered on the current location.

int/color **fillcanvas** –

Fills the entire current canvas with the gray value or color.

delta r x y w h **insetrrect** r′ x′ y′ w′ h′

Similar to **insetrect**, but with a rounded rectangle.

thickness x y w h **ovalframe** –

Similar to **rectframe**, but with an oval.

x y w h **ovalpath** –

Creates an oval path with the given bounding box.

thickness x y w h **rectframe** –

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

thickness r x y w h **rrectframe** –

Similar to **rectframe**, but with a rounded rectangle.

r x y w h **rrectpath** –

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

string **rshow** –

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

int/color **strokecanvas** –

Strokes the border of the canvas with a one point edge using the gray value or color. Currently only works for rectangular canvases.

5.3.5 Text and Font Utilities

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

array **cvas** string

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

int **cvis** string

Converts a (small) integer into a one character string.

string **findfilefont** font Reads a font family definition from the file named by *string*, and returns a unit-high font.

The font is entered into the FontDirectory under the font name in the family file. This provides a way to load a bitmap font into NeWS after it has started up.

font **fontascent** int

Returns *font*'s ascent.

font **fontdescent** int

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

font **fontheight** int

Returns *font*'s height.

string **stringbbox** x y w h

Returns *string*'s bounding box.

5.3.6 CID Utilities

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

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

id **cidinterest** interest

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

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

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

This procedure generates a unique id, then notifies the client to repair the **/Damaged** board by sending over a DAMAGE_TAG and MyCID. It then forks a process which listens for code fragments from the client to execute. The **waitprocess** waits for one of these fragments to exit the **forkeventmgr** callback loop.

id proc **sendcidevent** –

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

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

The *id* used is the one sent along with the DAMAGE_TAG. See Chapter 8, ''Mixing PostScript and C'' for an explanation of the use of **cdef**.

id **cidinterest1only** interest

A special form of **cidinterest** which processes only one code fragment. It automatically **exits** by itself, rather than requiring the client to send the exit. For example, the *go* demo uses this to respond to mouse buttons which place a single stone using the above drawing fragments.

5.3.7 Journalling Utilities

The following utilites allow you to control the journalling mechanism. With this mechanism it is possible to to record and play back 4Sight user input events. The file *$NEWSHOME/lib/NeWS/journal.ps* implements the following three procedures:

– **journalplay** –

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

– **journalrecord** –

Start a journalling session by opening the journalling file and logging user actions to it. The default filename is */tmp/NeWS.journal.*

– **journalend** –

Ends a journalling session started by **journalrecord** and closes the journalling file.

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

5.3.8 Journalling Internal Variables

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

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

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

5.3.9 Constants

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

– **console** file

Returns the file object for the system's console. Use with **fprintf** to write messages to the console.

– **framebuffer** canvas

Returns the root canvas.

– **nulldict** dict

Returns an empty, small dictionary.

– **nullproc** procedure

Returns a no-op procedure.

– **nullstring** string

Returns an empty string.

5.3.10 Colors as Dictionaries

ColorDict is a dictionary that contains named colors. It is implemented in */usr/NeWS/lib/NeWS/colors.ps, which is loaded by init.ps* at startup. The color names come from the RGB values in X.10V4. Here are some examples:

```
/Aquamarine             112 219 147 RGBcolor def
/MediumAquamarine       50 204 153 RGBcolor def
/Black                  0 0 0 RGBcolor def
/Blue                   0 0 255 RGBcolor def
/CadetBlue              95 159 159 RGBcolor def
/CornflowerBlue         66 66 111 RGBcolor def
/DarkSlateBlue          107 35 142 RGBcolor def
```

RGBcolor simply converts 0 – 255 color values into colors:

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

5.4 Logging Events

The file *eventlog.ps* defines a procedure to turn logging of event distribution on and off, and a dictionary of events which should be excluded from the log. 'Logging' means that a copy of each event is printed as it is taken out of the event queue for distribution. This is useful for debugging the server and clients using events heavily. It adds **eventlog** and **UnloggedEvents** to **systemdict**. The fields of the event which are printed are **Serial#, TimeStamp, Location, Name, Action, Canvas, Process, KeyState**, and **ClientData**. Here's a sample log message:

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

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

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

6. Classes

While other window systems provide specific strutures for menus, windows, scrollbars, and the like, NeWS takes a different, more flexible approach. NeWS provides the *facilities* to build these higher-level user interface tools without imposing its notion of what these tools must be. Think of NeWS as providing the window management ''kernel'' from which a user interface toolkit may easily be made.

4Sight's window manager, Max, consists of a small set of user interface packages written entirely in PostScript. The content and nature of these packages is described in Section 3 of this manual, ''Programming the IRIS Window Manager''.

The 4Sight window manager packages are implemented using an object-oriented programming byte object scheme that is quite similar to Smalltalk. This scheme has several advantages:

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

This chapter discusses NeWS's implementation of Smalltalk's class mechanism.

6.1 Packages and Classes

Many of the essential ideas in class-based systems are similar to the more traditional ''package-'' or ''module-based'' systems, with the following distinctions:

- Packages (modules) are replaced by *classes*.
- Procedures in packages are replaced by *methods* in classes.
- Creating package objects is replaced by creating new *instances* of a class.
- Package local and global variables are replaced by *class variables*.
- Object variables are replaced by *instance variables*.
- Classes are ordered into a hierarchy by *subclassing* a new class to a prior one, *inheriting* its methods, instance variables, and class variables.
- Methods are invoked by use of the **send** primitive. The term *message* is used for an invocation of a method with its arguments.
- There is a means of constructing classes that is absent in most languages' module creation.
- Two new concepts, the *self* and *super* pseudo-variables, are introduced. They are used in methods to refer to the object that sent the message and the method's superclass, respectively.

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

Unlike PostScript procedures, methods are *compiled* when a class is created. Currently this simply resolves self and super, and performs some minor optimizations.

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

6.2 Introduction to Classes

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

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

classname superclass instancevariables **classbegin** –

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

– **classend** classname dict

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

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

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

– **self** instance

Used as the target object with **send, self** refers to the instance that caused the current method to be invoked. It does *not* refer to the class the method is defined in. The **self** primitive can also be used anywhere to refer to the currently active instance.

– **super** instance

Used as the target object with **send, super** refers to the method being overriden by the current method. Unlike **self, super** cannot be used outside the context of **send.**

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

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

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

6.3 Example Class: 'Foo'

Now lets build a sample class, '**Foo**':

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

% class methods
     /new {  % - => - (Make a new 'Foo')
          /new super send begin
               /resettime self send
               currentdict
          end
     } def
     /printvars {     % - => - (Print current state)
          (...we got: Value=%, Time=%.\n) [Value Time] printf
     } def
     /changevalue {  % value => - (Change the value of 'Value')
          /Value exch def
     } def
     /resettime {     % - => - (Change 'Time' to the current time)
          /Time currenttime def
     } def
classend def
```

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

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

Let's look at some uses of '**Foo**.' Here we create a new instance, '**foo**' of '**Foo**.' We then print out its initial values, shown by the line starting with '...we got'.

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

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

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

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

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

Now we do an odd thing, we simply send an executable array (a procedure) to '**foo**.' The effect of doing this is to execute the procedure within the context of '**foo**.' The procedure we're sending to '**foo**' is **{/Time ClassTime def}**, which assigns '**ClassTime**,' the class variable, to '**Time**,' the instance variable.

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

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

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

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

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

7. Input Handling in NeWS

Input handling in NeWS is a major extension to PostScript. This chapter describes the primitive data structures and operations for handling input in NeWS. These primitives provide a minimal user-input system. More sophisticated user-interfaces can be constructed entirely in PostScript on top of these primitives. For information on such an interface, see Section 3 of this manual, ''Programming the IRIS Window Manager''.

7.1 Input Events

Input in NeWS is treated as a series of *events* that are received, translated, dispatched, and routed by the server to its PostScript clients. Events are structured objects. They contain a number of fields, which are accessed as though the event were a dictionary and the fields were keys in that dictionary. Most of these fields are mentioned as they become relevant in this chapter. The full definition of event fields can be found in Chapter 4, ''New PostScript Types''. The PostScript interface is structured in such a way that adding fields does not affect existing PostScript programs.

Among the most interesting fields in an event are:

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

Many events are generated by the system to report user actions like mouse motion and key presses. Events can also be generated by PostScript processes, and submitted for processing just like system-generated events. The **createevent** primitive leaves a new event on the stack with null or 0 in all fields. The **copy** primitive can be used to copy the fields of one event into another *en masse*. This extension of the **copy** primitive is closely analogous to its usage with two dictionaries. (For the integrity of the input system, the event's **IsInterest** flag and **Serial** number are not copied.)

7.2 Submitting Events for Distribution

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

sendevent takes an *event* off the stack. The event is sorted into the event queue and distributed as described in Section 7.4, ''Event Distribution''.

recallevent takes an *event* off the stack and removes that event from the event queue. This implies that the event must have been put in the queue using **sendevent**, since no process will have a reference yet for system-generated events. It also requires that the client must have saved some reference to the event given to **sendevent**, in order to pass the same event to **recallevent**. **recallevent** is useful for turning off a timer event that has been sent but not yet delivered.

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

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

7.3 Interests: Event Selection

Processes indicate their interest in receiving events by constructing one or more *ideal* events resembling what they want to receive and passing each ideal event as an argument to **expressinterest**. In other words, to get a particular kind of event, create an event like it and express interest in that event. An event that has been used as an argument to **expressinterest** in this way is called an *interest*. When real events are generated and distributed, they are are matched to interests on the **Name**, **Action**, **Canvas**, and **Process** fields. Non-specific (wild-card) matches may be specified, as described under Section 7.4.2, ''Event Matching''. Two other event fields that affect the matching behavior of a field are its **Exclusivity** and its **Priority**; their effects are described under Section 7.4.4, ''Order of Interest Matching''.

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

- they have never been passed to **expressinterest**,
- they have subsequently been passed to **revokeinterest**,
- the process that expressed the interest has terminated, or
- the canvas on which the interest was expressed is destroyed.

7.3.1 Changing and Reusing Interests

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

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

7.3.2 Inquiring for Current Interests

The set of interests that are currently active for any canvas or process may be retrieved as an arrqay of events, by treating the canvas or process as a dictionary and getting the value associated with the **Interests** key. This mechanism is described fully in Chapter 4, ''New PostSCript Types''. Similarly, the *gloabal interest list*, the set of interests that have been expressed with **null** in their **Canvas,** is returned by the operator **globalinterestlist**. The result is an array of events, ordered by priority (highest first).

7.4 Event Distribution

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

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

7.4.1 Receiving Events

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

7.4.2 Event Matching

Matching between a real event and an interest is defined below.

Name and Action Matching

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

Canvas Matching

A null **Canvas** matches events happening anywhere. A non-null canvas in the interest will match events occurring when the cursor is within that canvas, or other events with the canvas field set to that canvas (e.g., **/Damaged** events).

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

Process Matching

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

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

7.4.3 Processing After an Interest Match

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

- If the value in the dictionary corresponding to the matching key is not executable, then that value replaces the **Name** or **Action** field in the event.
- If the dictionary value *is* executable, then the value in the corresponding field of the event is not modified; instead, the executable object from the dictionary is queued for execution in the receiving process immediately after the event is returned by **awaitevent**.

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

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

7.4.4 Order of Interest Matching

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

Interest Lists

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

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

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

Multiple Matches

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

- Any interest may have its **Exclusivity** field set true; if so, an event that matches that interest will not be considered against any further interests.
- A canvas may absorb events, so that they are not tested against interests on any canvas behind it. This is controlled by the canvas' **EventsConsumed** field.

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

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

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

In these terms, the global interest list acts as if it were a list on a canvas that consumes **MatchedEvents**; events that match an interest on the global list never get through to any canvas-specific list unless they are specifically redistributed.

7.5 Special Event Types

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

- **/Damaged**: Damage events are generated for a canvas whenever it is damaged. By the time a process repairs the damage, several events may have accumulated. The total damage is accessible with **damagepath**. The **Action** for a damage event is null, and the **Canvas** field identifies the affected canvas.

- **/EnterEvent, /ExitEvent**: When the cursor is moved across a border between canvases, multiple events are generated. In each event, the **Name** is either **/EnterEvent** or **/ExitEvent**, depending on the direction of the crossing. Details of the **Action** are described in the next section.

- **/MouseDragged, /LeftMouseButton, /MiddleMouseButton, /RightMouseButton**: Manipulation of the mouse generates events with these names. If the mouse moves, the event **Name** is **/MouseDragged** and the **Action** is null. If a mouse button is pressed or released, the **Name** identifies which button is affected and the **Action** is one of the keywords **DownTransition** or **UpTransition**.

- Timer events: There are no special timer events in NeWS; rather, the guarantee that no event will be delivered from the event queue before the time in its **TimeStamp** means that any event can be used to generate another event at some time in the future. The example program at the end of this chapter illustrates a timer event.

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

7.5.1 Actions for Enter and Exit Events

Window-crossing events are generated whenever the cursor crosses the boundary between two canvases. These events are directed to the **Canvas** field of each event and specify how the cursor moves with respect to that canvas. The **Name** field of the event is either **/EnterEvent** or **/ExitEvent**. The **Action** field of the event is 0, 1, or 2. The definitions below help explain what these values mean.

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

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

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

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

Table 7-1. Boundary Crossing Events

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

7.6 Input Synchronization

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

In some cases, a stricter guarantee of synchronization than this is required. For instance, suppose one process sees a mouse button go down and forks a new process to display and handle the menu until the corresponding button-up. The new process must be given a chance to express its interest before the button-up is distributed, even if the user releases the button immediately. In general, whenever processing of one event may affect the distribution policy, distribution of the next event must be delayed until the policy change has been completed. This is done with the **blockinputqueue** primitive.

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

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

7.7 Example Program

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

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

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

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

- The **Name** must be one of '**/Tick**,' '**/Tock**,' or '**/Pumpkin**' (keys in the dictionary '**d**' in the **Name**).
- Any **Action** is valid (null in the **Action**).
- The location of the event doesn't matter, but events directed to a specific canvas will not match (null in the **Canvas**).

Because the **Name** in the interest is a dictionary, special processing is invoked on matching events. If the event **Name** is '**/Tick**' or '**/Tock**,' it will be translated to the other as the event is matched. (Non-executable values in the dictionary replace the value in event field.) If the event **Name** is '**/Pumpkin**,' it will not be changed; rather, the value in '**d**' of the procedure defined as '**/Pumpkin**' will be executed as the **awaitevent** at line 28 returns. (Executable values in the dictionary are queued for execution without changing the field in the event.)

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

Lines 20 – 24 similarly create and initialize another event, '**e3**,' and leave it on the stack for the main loop. The first part of the loop (lines 25 – 27) adds 1/60 of a minute to the **TimeStamp** of the event on the top of the stack (which is '**e3**' on the first time through), and sends it back for distribution a second later.

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

8. Mixing PostScript and C

The C to PostScript (CPS) interface has been designed to facilitate interactions between programs written in the C language on the client side and PostScript on the NeWS server side.

The interface model is of a client program which constructs a NeWS program and then, after opening a connection to the server, transmits the program. This program may make use of the all the built-in features of NeWS (including procedures already defined in the **userdict** and the **systemdict**).

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

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

8.1 How to Use CPS

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

- the *.cps* file that contains PostScript code executed by the server
- the *.h* file that contains PostScript code in a form recognizable to the C compiler
- the *.c* file, the client program, which can use PostScript programs.

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

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

```
% cps test.cps
```

The input file in the above example is translated by CPS into a header file (a *.h* file).

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

This header file is then *#include*'d in the client (C) program before compilation. It will contain not only the definitions that you have made but also a number of additional functions that CPS provides (see Section 8.6, ''CPS Utilities'').

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

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

```
% cc -I$NEWSHOME/include test.c /usr/NeWS/lib/libcps.a
```

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

8.1.1 The *.cps* File

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

The *.cps* file contains definitions of the following form:

```
cdef macro_name() procedure
```

macro_name is the name of the macro as you wish to label it within your client side program. *procedure* is the PostScript procedure that you wish to invoke. For example, the *ps_moveto()* procedure is specified this way:

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

CPS understands how to construct very efficient C code fragments that package and transmit PostScript fragments. The arguments to the C procedure are inserted where they are referenced in the PostScript fragment. The invocation

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

causes this PostScript fragment to be transmitted:

```
10 20 moveto
```

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

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

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

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

8.1.2 Argument Types

Just as in normal C procedure declarations, the parameters to CPS macros must be given types. The syntax for specifying a type is different: the type name appears preceding the parameter in the parameter list. The default type of these arguments is **int** (as in the C language).

For example, the previous definition of *ps_moveto()* is equivalent to:

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

Most of the types correspond directly to C types.

The following table lists the CPS argument types:

CPS Type	C Type
`int`	*int*
`float`	*float* or *double*
`string`	*char* *
`cstring`	*char* * with an accompanying count of the number of characters in the string. Such a parameter is actually two parameters: the pointer to the string and the count.
`fixed`	A fixed-point number represented as an integer with 16 bits after the binary point. See the description of *integer* in Table 13-2, *Implementation Limits* of the .
`token`	A special user-defined token. This is described in the section on user tokens.

Table 8-1. CPS Argument Types

8.1.3 The *.h* File

The header (*.h*) file is created by the CPS utility. It should be included in your client C program using the *#include* feature of the C pre-processer. In addition to the routines that have been defined in the *.cps* file, this file includes a number of pre-defined PostScript routines (listed in the final section of this chapter, ''CPS Utilities''.

8.1.4 The *.c* File

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

8.2 Communication with the Server

Communication with the server is handled by three CPS functions. These functions manage the low-level work of determining which server to connect to, establishing the connection, requesting execution of PostScript code, and severing the connection to the server cleanly. These functions are part of a body of functions that define the CPS interface and which are made available by the inclusion of the CPS library during compilation.

8.2.1 Opening a Connection

A connection to the NeWS server is establishing by calling the CPS function *ps_open_PostScript()*. This function returns a *PSFILE* pointer if a connection to the NeWS server is successfully established, otherwise a *0*. The function determines which server to connect to by examining the environment variable *NEWSSERVER* For a detailed explanation see Appendix E, ''Supporting NeWS Without C''.

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

8.2.2 Connection Files

Two *PSFILE* pointers, *PostScript* and *PostScriptInput,* are the conduits through which information flows between NeWS server and client programs. When the client writes to the NeWS server, it is writing on the file represented by the pointer, *PostScript*. When it (the client) reads output from the server, it is reading the file *PostScriptInput*. All operations on these *PSFILE* pointers are done using the *psio* package, not the standard I/O. See the *psio*(3) manual page for an explanation.

8.2.3 Buffered Output to the Server

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

8.2.4 Closing a Connection

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

8.2.5 Comments

The CPS comment convention is the same as the PostScript comment convention: everything from a % sign to the end of a line is a comment.

8.3 Tags, Tagprint, and Typedprint

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

Section 8.1.1, ''The *.cps* File'', explains how to call a PostScript function from the client side. This tagged packet method may be said to be the complementary half of that system.

8.3.1 Tags

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

- a label (*name*) with *args* that the client side program can use.
- an identifier (*tag*).
- the PostScript routine or code fragment to be associated with the label.
- a list of variables to receive the values in the reply (*results*).

The syntax of a specification is:

cdef *name* (*args*) *PostScript code* => *tag* (*results*)

There are three phases in the client execution of a CPS procedure:

1. transmitting the PostScript code
2. waiting for the return of the tagged reply
3. setting any result values from the reply

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

```
cdef execute()
   makewin
```

This will execute the PostScript routine *makewin* within the server when the *execute()* function is called from the C client.

```
#define BBOX_TAG 57
cdef ps_bbox(x0,y0,x1,y1) => BBOX_TAG (y1, x1, y0, x0)
        clippath pathbbox       % Find the bounding box of
                                        % the current clip.

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

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

8.3.2 Receiving Tagged Packets from NeWS

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

```
MENU_HIT_TAG tagprint
menuindex typedprint
```

Tagged packets will come back from the server to the client at times determined by the user's interaction, possibly intermixed with replies to requests like *ps_bbox*. The tags let the client side libraries sort out which replies go to which requests.

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

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

In this case *ps_menu_hit()* is a function, not a procedure. It tests to see if the first message on the received message queue is a menu hit. If it is, then it unpacks its arguments and return true, removing the tag and arguments from the queue. Otherwise it returns false, leaving the queue alone. If there is nothing in the received message queue, the function waits until something is received.

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

```
while (!psio_error(PostScriptInput) {
        if (ps_menu_hit(index))
                handle_menu_hit(index);
        else if (ps_character_typed(character))
                handle_typed_character(character);
        else if (ps_redraw_requested())
                handle_redraw();
        else {
                /* illegal tag; program bug */
        }
}
```

8.3.3 *tagprint* and *typedprint*

The **tagprint** statement sends the tag '*BBOX_TAG*' back to the client. It places the specified value on the input stream (the file *PostScriptInput*) from which the client side retrieves it. The C client has been waiting for such a tag and the subsequent '**typedprint**'s return the coordinates to the client. The '**typedprint**'s can return any variables of the types listed in the Table 8-1.

8.4 A Sample Tags Program

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

A Server Side Tags Program

```
%
% A very simple NeWS client
%

#define SET_GRAYS_TAG       1

cdef initialize()
    /starpath { % x y w h  =>  -  (make a star path)
        matrix currentmatrix 5 1 roll        % xfm x y w h
        4 2 roll translate scale                     % xfm
        .2 0 moveto .5 1 lineto .8 0 lineto          % xfm
        0 .65 lineto 1 .65 lineto closepath          % xfm
        setmatrix                                    % -
    } def
    /FillCanvasWithStar { % stargray fillgray => -
        fillcanvas setshade
        clippath pathbbox starpath fill
    } def
    /SetStarGrays { % stargray fillgray => -
        SET_GRAYS_TAG tagprint typedprint typedprint
    } def


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

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

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

A Client Side Tags Program

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

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

main()
{
   float stargray, fillgray;

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

8.5 Tokens and Tokenization

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

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

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

Several operators are defined by the CPS utility to allow you to add and retrieve tokens from the token list. When a token is added to the list, it is available whenever the token is found by the scanner in the input stream. It is frequently useful to add font objects to the token list and save the lookup time.

NeWS has a mechanism, supported by CPS, where a client program and the server can cooperatively agree on the definition of a user token.

The CPS declaration tells CPS that you want to transmit the user-defined token *black* in compressed form.

```
usertoken black
```

When *black* appears in following CPS definitions, the compressed token is used in the definition.

In order to establish the meaning of the token, the client has to talk to NeWS before the first use of the token. There are a number of procedures that the C program can call to do this:

ps_define_stack_token(*u*)

> Takes the value on the top of the PostScript stack in the server and defines it as the value of the token *u*. In future messages to PostScript, *u* is this value.

ps_define_value_token(*u*)

> Defines the user token *u* to be the same as the current value of the PostScript variable *u*. In future messages to PostScript, *u* is the value that the PostScript variable *u* had at the time *ps_define_value_token()* was called. Future changes to the value of the PostScript variable *u*, or its identity as determined by changes in variable scope, have no effect on the definition of the token.

ps_define_word_token(*u*)

> Defines the user token *u* to be the name of the PostScript variable *u*. In future messages to PostScript, *u* the PostScript variable *u*. This

binds the token *u* to the name *u*. When it is sent to PostScript, the name *u* is evaluated and its value is used.

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

8.6 CPS Utilities

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

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

Table 8-2. CPS Utilities

9. Debugging in NeWS

A primitive PostScript debugging package is available in NeWS. It allows you to set breakpoints and print to debugging output windows. It also has a simple facility for automatically causing breaks when errors are encountered. Because the debugger is written in PostScript, you can modify it to better fit your needs.

9.1 Starting the Debugger

Debugging code is an interactive process. In order to debug in NeWS you must create one or more interactive connections to the NeWS server and start the debugger on each one. You can connect to the server from any shell using the NeWS PostScript shell command, *psh*(1). To start up the NeWS debugger in a text window, follow these steps:

1. At the system prompt, enter the *psh* command.

   ```
   % psh
   ```

   You are now in the NeWS PostScript shell. This shell does not use a prompt; enter subsequent commands when the cursor returns to a new (empty) line.

2. Invoke a PostScript executive process. The NeWS server responds with a greeting.

   ```
   executive
   Welcome to 4Sight Version 1.4
   ```

3. Load and start the debugger. The NeWS server will confirm the installation.

```
(NeWS/debug.ps) run
dbgstart
Debugger installed.
```

Note: If you are debugging PostScript code that you are running directly from an executive, start a debugging executive in another *psh* connection. This prevents the debugger from trying to break to itself. Use the first executive to run the code being tested, and the second one to trap the errors.

9.2 The Debugging Environment

dbgstart forks a debugger process that is attached to the *psh* connection and "listens" for debugger-related events generated by client commands. (Actually, all client commands simply broadcast debugger events to these debugger daemons.) Any client command that causes printing will print in each debugging *psh* connection.

Because you are debugging in a multi-process environment, you have the problem of debugging several processes at one time. The solution is to have each debugging connection maintain a list of processes that are paused for debugging. This list is printed via the **dbglistbreaks** command below. It is also printed whenever a new break occurs. Any of the listed breaks can be *entered* using the **dbgenterbreak** command. This swaps the *psh* debugging context out and replaces it with the paused process. The context currently consists of the dict stack and operand stack.

9.3 Debugging Commands

Debugging commands fall into two categories: *client commands*, executed from client programs and *user commands*, issued by a debugging user. User commands are executed from the *psh* connection to the server, while client commands are inserted within the code being debugged.

9.3.1 Client Command Summary

name **dbgbreak** –	break and print pending breaks
formatstring argarray **dbgprintf** –	print formatted arguments

9.3.2 Client Command Details

The following are detailed descriptions of the client debugger commands.

name **dbgbreak** –

Causes the current client to pause, printing the pending breaks in all debugger connections. *name* is used as a label in the list to distinguish between breaks.

See also: **dbgbreakenter, dbgbreakexit.**

formatstring argarray **dbgprintf** –

Prints on each debugger connection, in **printf** style. If there are no debugger connections, it prints on the console.

```
(Testing: % % %\n) [1 2 3] dbgprintf
Testing: 1 2 3
```

See also: **dbgprintfenter, dbgprintfexit**

9.3.3 User Command Summary

name/[dict name] **dbgbreakenter** –	break named procedure as it starts
name/[dict name] **dbgbreakexit** –	break named procedure as it exits
arg clientproc **dbgcall** –	implicit version of **dbgcallbreak**
arg clientproc brknum **dbgcallbreak** –	execute *clientproc* with *arg* as data
– **dbgcontinue** –	continue process
breaknumber **dbgcontinuebreak** –	continue process of id *breaknumber*
– **dbgcopystack** –	copy stack to process
– **dbgenter** –	enter last process listed
breaknumber **dbgenterbreak** –	change debug environment
– **dbgexit** –	return from process to debugger
breaknumber **dbggetbreak** process	return process of id *breaknumber*
– **dbgkill** –	kill default process
breaknumber **dbgkillbreak** –	kill process of id *breaknumber*
– **dbglistbreaks** –	list all pending breakpoints
name/[dict name] hproc tproc **dbgmodifyproc** –	run *hproc* before procedure, *tproc* after
level index patch **dbgpatch** –	patch the implicit process
level index patch breaknumber **dbgpatchbreak** –	patch the process of id *breaknumber*
name/[dict name] fstring argarray **dbgprintfenter** –	make procedure do **dbgprintf** as it starts
name/[dict name] fstring argarray **dbgprintfexit** –	make procedure do **dbgprintf** as it exits
– **dbgstart** –	start debugger
– **dbgstop** –	stop debugger
– **dbgwhere** –	print execution stack
breaknumber **dbgwherebreak** –	print stack of process of id *breaknumber*

9.3.4 User Command Details

Most of the user-level debugger commands come in two forms: one that explicitly takes a breaknumber and one that does not.

- A command of the form *cmdname***break** expects an explicit breaknumber for its argument.

- A command of the form *cmdname* (without the **–break** suffix) uses an implicit breaknumber. This number is generally the currently entered break, or the last break in the list if there is no currently entered break.

The implicit form is most often used when there is only one break pending, or if the same breaknumber must be used repeatedly.

The following are detailed descriptions of the user-level debugger commands.

name/[dict name] **dbgbreakenter** –

Modify the named procedure to call **dbgbreak** just after starting. If *name* is an array, it is assumed to be a dict and the name of a procedure in the dict. Thus, to break when a window is made, type:

```
[DefaultWindow /new] dbgbreakenter
Break:/new from process(4050350, input_wait)
Currently pending breakpoints are:
    1: /new called from process(4050350, input_wait)
```

See also: **dbgbreak.**

name/[dict name] **dbgbreakexit** –

Modify the named procedure to call **dbgbreak** just before exiting.

See also: **dbgbreak**

arg clientproc **dbgcall** –

Implicit version of **dbgcallbreak**. The **dbgcopystack** command, for example, uses this mechanism:

```
[count copy pop]     % args
{begin clear ClientData aload pop end}    % args proc
dbgcall      % –
```

arg clientproc breaknumber **dbgcallbreak** –

Execute **clientproc** in the broken process with *arg* as data. The **clientproc** primitive will be executed (in the client environment) with the event on the stack, thus is responsible for popping it off.

– **dbgcontinue** –

Continues the currently entered process or the last process listed if no process is currently entered.

breaknumber **dbgcontinuebreak** –

Continues the process identified by *breaknumber.*

– **dbgcopystack** –

Copies the current operand stack to the process being debugged. This allows you to **dbgenter** a process, modify that copy of the operand stack, and copy it back to the process.

– **dbgenter** –

Enters the last process listed. See **dbgenterbreak** below.

breaknumber **dbgenterbreak** –

As far as possible, make this debug connection have the same execution environment as the process identified by *breaknumber.* Currently, this includes the operand stack and the dictionary stack. **dbgenterbreak** allows you to browse around in the given process' state. If **dbglistbreaks** is executed while within an entered process, the listing will indicate that process with a '=>' in the left margin.

```
3 dbgenterbreak
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
  =>3: /menubreak called from process(5177764, breakpoint)
```

– **dbgexit** –

Return to the debugger connection from whatever process you may have entered. This is a no-op if no process is currently entered. Most debugger commands will call this for you if appropriate.

breaknumber **dbggetbreak** process

Returns the PostScript process object for the given breaknumber.

– **dbgkill** –

Kills the default process.

breaknumber **dbgkillbreak** –

Kills a breakpointed process, removing it from the breaknumber list.

– **dbglistbreaks** –

List all the pending breakpoints resulting from **dbgbreak** above. They are listed in the following form:

```
dbglistbreaks
Currently pending breakpoints are:
    1: /oneA called from process(4245774, breakpoint)
    2: /oneB called from process(4306134, breakpoint)
    3: /menubreak called from process(5177764, breakpoint)
    4: /undefined called from process(4154624, breakpoint)
```

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

name/[dict name] headproc tailproc **dbgmodifyproc** –

Modify the named procedure to execute *headproc* just before calling it, and to call *tailproc* just after calling it. In effect, '**{proc}**' becomes '**{headproc proc tailproc}**.' This is the mechanism for implementing **dbgbreakenter/exit** and **dbgprintfenter/exit**.

level index patch **dbgpatch** –

Patch the implicit process.

level index patch breaknumber **dbgpatchbreak** –

Patch the execution stack for breaknumber process. The patch overwrites the word in the executable at the given level, and at the given index within that level. Prints the resulting execution stack (**dbgwhere**).

name/[dict name] formatstring argarray **dbgprintfenter** –

Modify the named procedure to call **dbgprintf** with *formatstring* and *argarray* just after starting. Note that *argarray* can be a literal array if you want to defer evaluation of the arguments until the **dbgprintf** occurs.

See also: **dbgprintf**.

name/[dict name] formatstring argarray **dbgprintfexit** –

Modify the named procedure to call **dbgprintf** with *formatstring* and *argarray* just before exiting. Note that *argarray* can be a literal array if you want to defer evaluation of the arguments until the **dbgprintf** occurs.

```
[DefaultWindow /reshape] (resize: % % % %\n)
{FrameX FrameY FrameWidth FrameHeight} dbgprintfexit
resize: 91 100 179 181
resize: 91 94 223 187
```

See also: **dbgprintf**.

– **dbgstart** –

Make the current connection to the server a debugger. Required before any of the other commands below can be used.

– **dbgstop** –

Removes the debugger from your *psh* connection.

– **dbgwhere** –

Prints the execution stack for the currently entered process or for the last process listed if no process is currently entered.

breaknumber **dbgwherebreak** –

Prints a exec stack trace for the process identified by *breaknumber.*

```
1 dbgwherebreak
Level 1
  { /foo 10 'def' /bar 20 'def' /A 'false' 'def'
    (Testing: %\n) 'mark' msg ] dbgprintf
    /oneB *dbgbreak } (*21,22)
Level 0
  { 100 'dict' 'begin' array{22} *'loop' 'end' } (*4,6)
```

The asterisk indicates the currently executing primitive in each level. The two numbers following each procedure are the index, relative to zero, of the asterisk and the size of the procedure. This is useful information for using **dbgpatch.**

9.4 Error Handling

When the debugger is running, a client error causes the client program to break to the debugger, exactly as if the command '/<*errorname*> **dbgbreak**' was inserted at the point in the client program where the error occurred. The following example illustrates the result of an **undefined** error occurring in a client while the debugger is running.

```
Break:/undefined from process(4154624, breakpoint)
Currently pending breakpoints are:
    1: /undefined called from process(4154624, breakpoint)
```

See the *PostScript Language Reference Manual* for details on error handling.

9.5 Aliases

Because the debugger is PostScript-based, the debugger commands can be easily modified. One common change is to define some easily-typed aliases for the verbose command names. Adding the code below to your *user.ps* file will make the aliases available in all debugging connections.

```
/dbe{dbgbreakenter} def
/dbx{dbgbreakexit} def
/dc {dbgcontinue} def
/dcb{dbgcontinuebreak} def
/dcc{dbgcopystack dbgcontinue} def
/dcs{dbgcopystack} def
/de {dbgenter} def
/deb{dbgenterbreak} def
/dgb{dbggetbreak} def
/dk {dbgkill} def
/dkb{dbgkillbreak} def
/dlb{dbglistbreaks} def
/dmp{dbgmodifyproc} def
/dp {dbgpatch} def
/dpe{dbgprintfenter} def
/dpx{dbgprintfexit} def
/dw {dbgwhere} def
/dwb{dbgwherebreak} def
/dx {dbgexit} def
```

Appendix A: Implementation Notes

A.1 Operator Omissions

The following PostScript primitives were defined by Adobe, but have not yet been implemented in the NeWS PostScript interpreter.

Primitive	Note
banddevice	Printer specific.
charpath	Not fully implemented.
copypage	Not fully implemented.
currentscreen	
currenttransfer	
echo	Printer specific.
executeonly	
framedevice	Printer specific.
invertmatrix	
noaccess	
nulldevice	Printer specific.
prompt	Printer specific.

Table A-1. Omitted PostScript Primitives

Primitive	Note
renderbands	Printer specific.
resetfile	
restore	Not fully implemented.
reversepath	
save	Not fully implemented.
setcharwidth	Part of the full font model.
setscreen	
settransfer	
showpage	Not fully implemented.
start	Replaced by *user.ps* initialization.
translate	Missing matrix argument version.
usertime	

Table A-1. (continued) Omitted PostScript Primitives

A.2 Implementation Limits

The following implementation limits exist in 4Sight version 1.3.

Quantity	Limit	Explanation
integer	32767	Integers are represented as 32 bits, 16 bits of them fraction. Integers are automatically converted to reals if they overflow.
real		Single-precision floating-point numbers are used. Reals are represented as fractional integers if they are small enough, but the type determination operators will describe them as real.
array	32767	Number of entries in an array.
dictionary	16384	Number of key/value pairs in a dictionary.
string	32767	Number of characters in a string.
name	32767	Number of characters in a name.
file	93	Number of open files (includes open client communication channels).
userdict	100	Set by code in *init.ps*; easy to change.
operand stack	1500	Maximum size of an operand stack.

Table A-2. Implementation Limits

Quantity	Limit	Explanation
dict stack		Expanded as required.
exec stack	100	Maximum function/compound statement nesting depth.
gsave level		Expanded as required.
path		Expanded as required.
VM		The server expands to use as much VM as the underlying system permits.
interpreter level		Not applicable.
save level		Not applicable.

Table A-2. (continued) Implementation Limits

Appendix B: Autobind

When the PostScript interpreter encounters an executable name, the interpreter searches the dictionary stack from the top to the bottom until it finds a definition for this name. The execution time will therefore increase as the size of the dictionary stack increases. On the other hand, this method allows one to redefine the behavior of a name by defining it in a dictionary and placing this dictionary on the dictionary stack.

PostScript provides an operator called **bind** that will circumvent this name lookup process. **bind** goes through a procedure and checks each executable name inside it. If a name resolves to an operator object in the context of the current dictionary stack, then **bind** alters the procedure by replacing the name with the operator object. This eliminates the time taken by name lookups when executing this procedure, but it removes the flexibility of being able to change a procedure's behavior by redefining names before executing it.

NeWS implements an *autobind* mechanism which will cause every executable procedure to behave similar to the way it would as if **bind** had been called on it. However, there is a subtle difference between the autobind mechanism and the **bind** operator. When the autobind mechanism checks to see if a name resolves to an operator, it looks only in **systemdict**, not in the dictionary stack.

The following example illustrates the differences among the cases with no binding, using **bind**, and using autobinding.

```
% psh
executive
Welcome to NeWS version 1.0
false setautobind
/test1 { 5 3 add == } def
/test2 { 5 3 add == } bind def
/add { sub } def
/test3 { 5 3 add == } bind def
true setautobind
/test4 { 5 3 add == } def

test1
2
test2
8
test3
2
test4
8
```

In this example, the '**test1**' procedure calls '**add.**' Since '**add**' was redefined to be '**{ sub }**,' '**test1**' really does a subtraction. The **bind** operator was used on procedure '**test2,**' so the '**add**' was not redefined as it was for '**test1.**' **bind** was run on '**test3,**' but since '**add**' now resolves to something other than an operator, no binding takes place. The '**test4**' procedure was defined with autobind turned on, so binding takes place, in spite of the redefinition of '**add.**'

Autobinding is on by default. Autobinding can be turned on or off using the **setautobind** operator, which takes a boolean argument. You can use the **currentautobind** operator to get the current setting; it returns a boolean value. If you want to redefine the behavior of a name that is defined in **systemdict**, you should make sure that autobinding is off when the name is redefined and when procedures that use the new definition are defined.

Appendix C: Byte Stream Format

The information in this section is only of interest to those implementing the NeWS protocol. Most C programmers should use CPS, which deals with all of the protocol issues transparently.

The communication path between NeWS and a client is a byte stream that contains PostScript programs. The basic encoding, which is compatible with PostScript printers, is simply a stream of ASCII characters. NeWS also supports a compressed binary encoding which may be freely intermixed with the ASCII encoding. The two encodings are differentiated based on the top bit of the eight-bit bytes in the stream. If the top bit is zero, then the byte is an ASCII character. If it is one, then the byte is a compressed token. This differentiation is not applied within string constants or the parameter bytes of a compressed token.

C.1 Encoding

Each compressed token is a single byte with the top bit set. There may be parameter bytes following it and there may be a parameter encoded in the bottom bits of the code byte. In the following description of the various tokens, the values are referred to symbolically. The mapping between these names and numeric values is given at the end of this appendix.

enc_int

*enc_int+(d<<2)+w; w*N*

0<=w<=3 and 0<=d<=3: The next *w+1* bytes form a signed integer taken from high order to low order. The bottom *d* bytes are after the binary point. This is used for encoding integers and fixed point numbers.

enc_short_string

*enc_short_string+w; w*C*
0<=w<=15: The next *w* bytes are taken as a string.

enc_string

*enc_string+w; w*L; l*C*
0<=w<=3: The next *w+1* bytes form an unsigned integer taken from high order to low order. Call this value *l*. The next *l* bytes are taken as a string.

enc_syscommon

enc_syscommon+k
0<=k<=32: Inside the NeWS server there is table of PostScript objects. The **enc_syscommon** token causes the *k*th table entry to be inserted in the input stream. Typically these names are primitive PostScript operator objects. This table is a constant for all instances of PostScript the contents of the table are 'well-known' and static. This token allows common PostScript operators to be encoded as a single byte.

enc_syscommon2

enc_syscommon2; k
0<=k<=255: This is essentially identical to **enc_syscommon** except that the index into the object table is *k+32*. This allows the less common PostScript operators to be encoded as two bytes.

enc_usercommon

enc_usercommon+k
0<=k<=31: This is similar to **enc_syscommon** except that it provides user-definable tokens. Each communication channel to the server has an associated PostScript object table. The **enc_usercommon** token causes the *k*th table entry to be inserted in the input stream. The table is dynamic; it is the responsibility of the client program to load objects into this table. The PostScript operator **setfileinputtoken** associates an object with a table slot for an input channel.

enc_lusercommon

enc_lusercommon+j; k
0<=j<=3 and 0<=k<=255: This is essentially identical to **enc_usercommon** except that the index is *(j<<8)+k*.

enc_IEEEfloat

*enc_IEEEfloat; 4*F*

The next four bytes, high order to low order, form an IEEE format floating-point number.

enc_IEEEdouble

*enc_IEEEdouble; 8*F*

The next eight bytes, high order to low order, form an IEEE double precision floating-point number.

C.2 Object Tables

The **enc_*common*** tokens all interpolate values from object tables. The appearance of one of these tokens causes the appropriate object table entry to be used as the value of the token. These tokens are typically a part of a PostScript stream that is executed and can be any kind of PostScript object. Usually either executable keyword or operator objects are used.

This has some subtle implications with scope rules. If the object is a keyword, then its value is looked up before being executed, just as an ASCII encoded keyword is. If it is an operator object, then the operator is executed directly, with no name lookup. This improves performance, but it also binds the interpretation of the object table slot at the time that the slot is loaded.

For example, if the executable keyword **moveto** is loaded into a slot, then whenever that token is encountered, **moveto** is looked up and executed. On the other hand, if the value of **moveto** is loaded into the slot, then whenever that token is encountered, the interpretation of **moveto** at the time the slot was loaded is used.

Table C-1 shows the bindings between token names and values.

Value	Span	Symbolic Name
0200	16	enc_int+(d<<2)+w
0220	16	enc_short_string+w
0240	4	enc_string
0244	1	enc_IEEEfloat
0245	1	enc_IEEEdouble
0246	1	enc_syscommon2
0247	4	enc_lusercommon
0253	5	free
0260	32	enc_syscommon
0320	32	enc_usercommon
0360	16	free

Table C-1. Token Values

C.3 Examples

The PostScript code fragment:

```
10 300 moveto
(Hello world) show
```

can be encoded simply as an ASCII text string:

```
"10 300 moveto\n(Hello world) show"
```

which gives a message that is 33 bytes long. The space following **show** is a delimiter; without it the tokens would run together. Binary tokens are self-delimiting. If the tokens were sent in compressed binary format then the message is sent in 19 bytes; Table C-2 explains what each byte represents.

Byte	Meaning
0200	encoded integer, one byte long, no fractional bytes
0012	the number 10
0201	encoded integer, two bytes long, no fractional bytes
0001	first byte of the number 300
0054	second byte of the integer, (1<<8)+054==0454==300
0261	**moveto** — assuming that **moveto** is in slot one of the operator table, which it isn't
0233	(0220+11) start of an 11-character string
0110	'H'
0145	'e'
. . .	
0144	'd'
0262	**show** — assuming that **show** is in slot two of the operator table, which it isn't

Table C-2. Meaning of Bytes in Encoding Example

Appendix D: Font Tools

4Sight manages fonts for NeWS, GL, and DGL clients with a single Font Manager. The IRIS Font Manager replaces the NeWS font management scheme, but provides the same functionality within NeWS, and is transparent to NeWS client programs. See Section 1 of this manual, ''Using the GL/DGL Interfaces'' for detailed information on the IRIS Font Manager.

This appendix describes how to convert fonts purchased from a third party into the Font Manager format and how to place them in the Font Manager's font library. It also describes how to create simple bitmap fonts for cursors and icons.

D.1 Building Text Fonts

The IRIS Font Manager supplies a set of standard fonts accessible to all window manager clients. These fonts are stored in a binary format, and are readily accessible. This section describes how to add new fonts to the set of existing Font Manager fonts.

The Font Manager is loosely modelled on PostScript. In PostScript, a font like **Times-Roman** is an object that can be scaled. A font file containing a set of bitmaps is an instance of a PostScript font at some particular size and orientation. A group of these font files, is called a *family,* and can be used to implement a PostScript font.

There are two steps to create a PostScript font from a set of font files:

1. The font files must first be converted into Font Manager format using *dumpfont.*
2. A description of them as a family must be built using *bldfamily.*

dumpfont(1) will take a set of Adobe ASCII Binary Distribution Format or *vfont* format files and convert them to Font Manager format with file extension *.fm*. *bldfamily*(1) will take a set of Font Manager fonts and build a font family file with extension **.ff**. For a full description of these programs and their options, see their man pages.

For example, say you have a set of Adobe *BDF* files *gachab7.bdf*, *gachab10.bdf*, *gachab12.bdf*, and *gachab14.bdf*; and you would like them to appear in NeWS and the Font Manager as the PostScript font **Gacha-Bold**. First, remove any old font information generated for the 4Sight servers in the working directory (not from the Font Manager font directory):

```
% rm -f fmaliases
```

Then, call *dumpfont* with this command line:

```
% dumpfont -fm -n Gacha-Bold gachab*.bdf
```

dumpfont will convert the files named *gachab*.bdf* into Font Manager format, write them into the current directory, calculate their size information by inspecting the bitmaps, and force the name to **Gacha-Bold**. *dumpfont* truncates the name of the output file names, using the font name as its start, not the name of the *dumpfont* input file.

Next, call *bldfamily* with this command line:

```
% bldfamily GachaB
```

Note that the truncated name of the **.fm** files is used. Note also that *bldfamily* may truncate the family file name differently, since it does not preserve size information in the file name. *bldfamily* will scan the current directory for files named *GachaB*n*.fm*. It will then build a font family file and write it to *GachaB.ff*.

After this is done, copy the family file and the font data files to the font directory:

```
% cp GachaB*.fm GachaB.ff /usr/lib/fmfonts
```

dumpfont appends newly-generated font family names to the file *fmaliases* in the directory in which the family was created. Append this file to the master *fmaliases* file in the font directory:

```
% cat fmaliases >> /usr/lib/fmfonts/fmaliases
```

If you bought an Adobe Binary Distribution Format *widths* file also, you should convert that to Font Manager format using *dumpfont* and copy it into the Font Manager directory:

```
% dumpfont -fm GachaB.afm
% cp GachaB.fw /usr/lib/fmfonts
```

A *widths* file contains a table of ideal printer widths the Font Manager can use in scaling and rotation.

You can now use the following PostScript code in a NeWS client program to pick up the font family that you have built:

```
(Gacha-Bold) findfont
```

You can use the PostScript **scalefont** primitive to select from the different sized bitmaps; the Font Manager will provide the closest size bitmap available. In some cases, it will generate a new font from existing data.

If you are running a GL client, you may access the new fonts through Graphics Library routines. See Section 1 of this manual, ''Using the GL/DGL Interfaces'' for more information.

D.2 Cursor Fonts

NeWS supplies the beginnings of a standard cursor font. The files involved are:

```
/usr/NeWS/lib/NeWS/cursor.ps
/usr/lib/fmfonts/Cursor.ffam
/usr/lib/fmfonts/Cursor12.font
```

The advantages of using the standard cursor font involve a more uniform look between applications and greater resource sharing within the window manager.

Note that this section describes how to *replace* your cursor font. You cannot add a glyph to the shipped cursor font without obtaining the image data for those glyphs from Silicon Graphics. This data is not part of the distribution because of its size.

D.2.1 Cursor Representation

At present, The IRIS Font Manager does not implement the full PostScript font mechanism. It supports only bitmap fonts. The closest approximation to the requested font will be selected from the font library of bitmaps.

Thus, cursor font shape descriptions are just bitmaps. The primary image is rendered in black over a white mask image. There is no such thing as an XOR cursor.

D.2.2 Cursor Format

The font utility *mkiconfont* expects input in the format illustrated by the examples below. Here is an example of a cursor named *pointer* that is used for the root window in the default *init.ps* file. Its image is that of a narrow arrow that points up and to the left.

```
/* Format_version=1, Width=16, Height=16, Depth=1,
 * Valid_bits_per_item=16
 * XOrigin=0, YOrigin=15
 */

0x0000,0x4000,0x6000,0x7000,0x7800,0x7C00,0x7E00,0x7800,
0x4C00,0x0C00,0x0600,0x0600,0x0300,0x0300,0x0180,0x0000
```

XOrigin and **YOrigin** indicate the origin of the character, which is the hot-spot of the cursor. The values for **XOrigin** and **YOrigin** originate in the bitmap's lower left-hand corner with positive values extending up and to the right. **YOrigin** is strange in that it starts from the last non-zero row of pixels, not the bottom of the bitmap.

Note: *mkiconfont* uses Sun Pixrect coordinate space. The output of *mkiconfont* must be translated into Font Manager format using *dumpfont*, as described below.

Here is another example of a cursor named *right_arrow*. Its image is that of an arrow that points right.

```
/* Format_version=1, Width=16, Height=16, Depth=1,
 * Valid_bits_per_item=16
 * XOrigin=17, YOrigin=6
 */

0x0000,0x0020,0x0030,0x0038,0x003C,0x7FFE,0x7FFF,0x7FFE,
0x003C,0x0038,0x0030,0x0020,0x0000,0x0000,0x0000,0x0000
```

It is OK for the origin of the character to be off the edge of the bitmap.

Cursors have a mask image and a primary image. Here is the mask for the *pointer* cursor. It is called *pointer_mask.*

```
/* Format_version=1, Width=16, Height=16, Depth=1,
 * Valid_bits_per_item=16, XOrigin=0, YOrigin=16
 */
    0xC000,0xE000,0xF000,0xF800,0xFC00,0xFE00,0xFF00,0xFF80,
    0xFE00,0xDF00,0x9F00,0x0F80,0x0F80,0x07C0,0x07C0,0x03C0
```

The mask image is used to outline the primary image and thus its origin is offset by one from the primary image so as to superimpose the images correctly. This is typical of cursor masks.

D.2.3 Generating a Cursor Font

Here is the process for generating a simple cursor font:

1. Generate a collection of ASCII bitmap file pairs (see the *Format* section above). Follow these naming conventions for the cursor and cursor-mask files:

   name`.cursor`
   name`_mask.cursor`

   Create a file containing these file names, each name on a separate line. In this example, the file is called *myfont.list* — you can name it whatever you want. The pair order should be primary file name followed by mask file name.

2. Make an ASCII version of the font from the list of ASCII bitmap files using the program *mkiconfont*. The first argument to *mkiconfont* is a file

containing a list of file names. The second argument to *mkiconfont* is the name of the output file prepended by a greater-than sign (>) and the intended name of the font family.

```
% mkiconfont myfont.list MyFont>MyFont12.afont
```

3. Turn the ASCII version of the font into a binary version using the program *dumpfont*. The argument is the name of the file of the ASCII version of the font. The output file is named like the ASCII version but with a *.fm* suffix instead of a *.afont* suffix. The output file is written to the current directory.

```
% dumpfont -fm MyFont12.afont
```

4. Build a font family file for the font using the program *bldfamily*. The **-d** argument causes the output file to be placed in the current directory. *bldfamily* looks in the current directory to find the font files that start with the family name. These font files are used as additional input to *bldfamily*.

```
% bldfamily MyFont
```

5. To reference the font symbolically from NeWS, you can build a *.ps* file that contains a dictionary of character names for the font. Here is an example of the way to do this:

```
#! /bin/sh
egrep "^(STARTCHAR|ENCODING)" MyFont12.afont>myfont.ps
ed - myfont.ps<<'EOF'
g/STARTCHAR/j
1,$s'STARTCHAR *)ENCODING *)'/1 /2 def'
1i
/myfontdict 300 dict def
myfontdict begin
$a
end
% Usage: x y moveto /myfontname showmyfont
/showmyfont {
        currentfont ( ) dup 0 myfontdict 5 index get put
        myfontfont setfont show setfont pop } def
/myfont (MyFont) findfont 12 scalefont def
w
q
EOT
```

6. Copy the font and font family files to the font directory.

   ```
   % cp MyFont.ff MyFont12.fm /usr/lib/fmfonts
   ```

7. Copy the *.ps* file to a well known place.

   ```
   % cp myfont.ps /usr/NeWS/lib/NeWS
   ```

8. Add code to the beginning of your PostScript program to load the font in the *.ps* file.

   ```
   (NeWS/myfont.ps) run
   myfontdict begin
   myfont name name_mask setcanvascursor
   end
   ```

Appendix E: Supporting NeWS Without C

As it comes out of the box, the only language that is supported for NeWS clients (besides raw PostScript) is C. The CPS preprocessor is primarily responsible for providing C support. What CPS and the *libcps.a* library provide is a mechanism for contacting the server (**ps_open_PostScript()**) and a mechanism for creating and sending messages to NeWS on that I/O channel. The *psh*(1) and *say*(1) programs provide these mechanisms to users who want to send PostScript programs to the server.

E.1 Contacting the Server

To contact the server from a IRIX environment you first need to get the environment variable *NEWSSERVER*. This contains a string like the following:

```
3227656822.2000;mymachine.
```

The first number is the 32-bit IP address of the server in host byte order. The second number is its IP port number. You need to create a socket and connect it to this IP address and port. Following the semicolon in *NEWSSERVER* is the text name of the host on which the server is running, which you can ignore. The *setnewshost*(1) command is a shell script that fabricates the appropriate string for *NEWSSERVER*.

Once a connection has been established, all you need to do is write bytes down the stream as described in Appendix C, ''Byte Stream Format''. Remember that you don't need to use the compressed binary tokens, they are merely an optimization. It is perfectly satisfactory to send ASCII PostScript code with no compression.

E.2 Communicating with the Server

Eventually, a CPS-like program that is appropriate for the language should be written. The basis for such a program would be the input and output facilities that CPS uses; the program could write routines that called them, or macros that expanded into invocations of them, or whatever other technique suited the host language.

There is a C function called **pprintf()** which is the runtime output work-horse behind CPS. It is invoked in a manner identical to *fprintf()*(3s), with a format string that is interpreted in the same way. When values are output with **%s** or **%d** or any of the other formatting specifiers, they are output as compressed binary tokens. The rest of the format string is output as is; it may contain compressed tokens or simple ASCII.

Input from NeWS to the client appears as bytes that can be read from the server I/O stream. The format of these bytes is entirely up to the PostScript code downloaded by the client into the server, so it may be as simple or as complex as you wish. There is a facility in PostScript for writing objects back to the clients using the same compressed binary format as the client uses to send to the server and a corresponding C procedure *pscanf()* for interpreting these messages.

Appendix F: Class *LiteItem*

This chapter presents a class-based items package, called **LiteItem**. Items are simple, graphical input controls. The item package is a further demonstration of the use of classes and packages besides the window and menu packages described in Section 3 of this manual, ‘‘Programming the IRIS Window Manager’’.

The item package is used by the *itemdemo* demo program. The PostScript code for the items package is in the file *liteitem.ps* in */usr/NeWS/lib/NeWS*. The item package currently implements the base class, **Item** the subclass **LabeledItem** and several practical subclasses of **LabeledItem.**

liteitem.ps is not loaded by the NeWS server at startup, so be sure to start any program implementing items with the following code:

```
systemdict /item known not { (/usr/NeWS/lib/NeWS/liteitem.ps) run } if
```

F.1 Class *Item*

A common need in interactive systems is a simple, user-definable, graphic, interactive, input/output object. Examples are buttons, sliders, scrollbars, dials, text fields, message areas, and the like. The class **Item** defines a skeleton for such an object.

An item has these major components:

- A canvas that depicts the item and is the target of the item’s input.
- A set of procedures that paint the canvas and handle activation and tracking events.

- A current value and a procedure that notifies the client when that value changes due to action of the tracking procedures.
- Methods for creating, moving and painting the item, and for returning the item's location and bounding box.

There are two utilities for items that reside outside of the class itself: **forkitems** and **paintitems**:

items **forkitems** process

Takes an array or dictionary of items and launches a process looking for an activation event (generally a mouse down event) for each of the items. When this occurs, a second event manager is forked that looks for events this particular item is interested in.

items **paintitems** –

Sends the **/paint** message to each of the items in an array or dictionary of items.

Let's take a look at the definition of class **Item**:

```
/Item Object [      % instance variables
    /ItemWidth    % item's width,
    /ItemHeight   % ...and height,
    /ItemParent   % ...and parent canvas (from new)
    /ItemCanvas   % the canvas we created for the item
    /ItemValue    % the canvas' current value
    /ItemInitialValue       % the value it started out with
    /ItemPaintedValue       % the value it currently shows
    /StartInterest    % the interest which activates the item
    /ItemInterests    % interests used to track item
    /ItemEventMgr     % ...the tracking process
    /NotifyUser   % the user's notify proc
] classbegin
% default variables
    /ItemFont           DefaultFont def
    /ItemTextColor      0 0 0 rgbcolor def
    /ItemBorderColor    ItemTextColor def
    /ItemFillColor      1 1 1 rgbcolor def
% class variables; mainly the std client procs
    /PaintItem    nullproc def      % the core of the /paint method
    /ClientDown   nullproc def      % procedures installed in
    /ClientDrag   nullproc def      %  the activated (tracking)
    /ClientEnter  nullproc def      %  process
    /ClientExit   nullproc def
    /ClientKeys   nullproc def
    /ClientUp     nullproc def
    /StopOnUp?    true def      % deactivate on up event?
```

```
% methods
    /new% parentcanvas width height => instance
    /makecanvas % - => -
    /makeinterests  % - => -
    /move   % x y => -  (Moves item to x y)
    /moveinteractive % item's backgroundcolor => -
    % (interactively moves the item)
    /paint  % - => -  ([Re]paints item)
    /location   % - => x y
    /bbox   % - => x y w h
classend def
```

The canvas and its "looks" are defined by **ItemWidth, ItemHeight, ItemParent, ItemCanvas, ItemFont, ItemTextColor, ItemBorderColor,** and **ItemFillColor.**

The parent canvas, height, and width are specified by the **/new** method. The others are initialized by the class and may be changed by the programmer or user.

The set of procedures for painting the canvas and handling activation and tracking events are **PaintItem, StartInterest, ItemInterests, ItemEventMgr ClientDown, ClientDrag, ClientEnter, ClientExit, ClientKeys, ClientUp,** and **StopOnUp?**

The **PaintItem** procedure is called by the **/paintitem** method, after it sets the canvas and does some minor bookkeeping. **StartInterest** is an event used by the *forkitems* utility to determine when to fork a second event manager, **ItemEventMgr**, to perform "tracking" of the item. **StartInterest** defaults to a mouse down event and generally is not overridden. **ItemInterests** is a dictionary of events used to track the item. It defaults to a set of events determined by which of the '*ClientFoo*' procedures have been overridden to be non-null. It is made by the **/makeinterests** method, which is generally invoked by the **/move** method as part of the item's deferred initialization. Clients are free to call **ItemInterests** themselves, however, if the need arises. **ItemEventMgr** is the tracking process and is null when tracking is not being performed. **StopOnUp?** is a boolean (default = true) that tells the tracking process whether to terminate on an up mouse event. (At present, only text items do not terminate on an up mouse event.)

The current value and notification procedures use **ItemValue, ItemInitialValue, ItemPaintedValue,** and **NotifyUser.**

ItemValue is the current value of the item. For example, it might be the string currently in a type-in item or true/false for a button item (indicating whether the button is currently on or off). **ItemInitialValue** is set to **ItemValue** when the item is activated for tracking. **ItemPaintedValue** is set to the value currently painted. These last two values are used to maintain a simple state machine by class implementations. **NotifyUser** is a procedure used to alert the client of changes in state.

F.1.1 Two Sample Items

These two subclasses of **Item** implement a simple toggle button and a simple slider. They both override the **/new** method, adding the initial **ItemValue** and the **NotifyUser** procedure to the argument list. Notice the way overriding is done:

```
/new super send begin
    ...
    currentdict
end
```

This is a standard PostScript programming style.

'*SampleToggle*' provides tracking by implementing the client '*Down*', '*Up*', '*Enter*', and '*Exit*' procedures. **ItemValue** is treated as a boolean, with true meaning "on". '*Down*' and '*Enter*' simply assign **not ItemInitialValue** to **ItemValue**, while '*Exit*' resets it to **ItemInitialValue.** '*Up*' simply calls the '*notify*' procedure if the state has changed. '*SampleToggle*' adds no instance or class variables.

```
/SampleToggle Item []
classbegin
    /new {  % initialvalue notifyproc parent width height => item
        /new super send begin
            /NotifyUser exch cvx def
            /ItemValue exch def
            currentdict
        end
    } def

    /PaintItem {
        ItemValue
            {0 fillcanvas}
            {1 fillcanvas 0 strokecanvas} ifelse
    } def
    /ClientDown {ItemInitialValue not SetToggleValue} def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientEnter {ClientDown} def
    /ClientExit {ItemInitialValue SetToggleValue} def
    /SetToggleValue {    % value => - (set value & paint toggle)
        /ItemValue exch store
        /paint self send
    } def
classend def
```

The '*SampleSlider*' provides tracking by implementing the client '*Down*', '*Up*', and '*Drag*' procedures. The '*Down*' and '*Drag*' procedures are identical, simply projecting the current *x* coordinate of the mouse onto the slider.

```
/SampleSlider Item []
classbegin
    /new { % initialvalue notifyproc parent width height => item
        /new super send begin
           /NotifyUser exch cvx def
        /ItemValue exch def
           currentdict
        end
    } def
    /PaintItem {
        ItemCanvas setcanvas 1 fillcanvas 0 strokecanvas
        ItemValue .5 PaintSliderValue
    } def
    /ClientDown {
        SetSliderValue
        ItemValue ItemPaintedValue ne {
            ItemPaintedValue 1 PaintSliderValue
            ItemValue .5 PaintSliderValue
        } if
    } def
    /ClientUp {ItemValue ItemInitialValue ne {NotifyUser} if} def
    /ClientDrag {ClientDown} def
    /PaintSliderValue { % value gray => -
        setgray
    2 sub 5 4 ItemHeight 8 sub rectpath fill
        /ItemPaintedValue ItemValue store
    } def
    /SetSliderValue {
        /ItemValue
            CurrentEvent /XLocation get
            5 max ItemWidth 5 sub min store
    } def
classend def
```

F.1.2 Sample Items Test Program

Now we'll play with these procedures using a simple test program. The '*notify*' procedure simply prints the value of the item using the **printf** utility. We start by creating a window and filling it's background. Then we make two items, a button and a slider, putting them in a dictionary called '*items*.' We then paint them and fork an activation event manager, '*p1*', that uses the middle mouse button to move the items interactively with the '*slideitem*' procedure. (This is a poor man's item editor that often is useful.)

```
/ItemSampleTest {
   /notify {(ItemValue: % 0 [ItemValue] /printf} def
   /FillColor .75 def
   /win {
      /FrameLabel (Sample Item Test) def
      /PaintClient {FillColor fillcanvas items paintitems} def
   } makewindowfromuser def
   /can win /ClientCanvas get def

    /items 50 dict dup begin
        /sampletoggle
            false /notify can 30 30 /new SampleToggle send def
            10 30 /move sampletoggle send
        /sampleslider
            90 /notify can 180 20 /new SampleSlider send def
            10 70 /move sampleslider send
    end def

   /slideitem { % items fillcolor item => -
      gsave
      dup 4 1 roll      % item items fillcolor item
      /moveinteractive exch send % item
      /bbox exch send        % x y w h
      grestore
   } def

   /p1 [
    items { % key item
        exch pop dup /ItemCanvas get % item can
        MiddleMouseButton [items FillColor % i c name dict color
        6 -1 roll /slideitem cvx] cvx    % can name proc
        DownTransition          % can name proc action
        4 -1 roll eventmgrinterest      % interest
        } forall
    ] forkeventmgr def
    /p1 items forkitems def
} def
```

F.2 Class *LabeledItem*

Most items are more elaborate than the preceding examples. **Class LabeledItem** implements a more common item; one that has a label-object pair, and an optional frame. The (abbreviated) class definition is:

```
/LabeledItem Item
dictbegin
% instance variables
    /ItemObject      nullstring def   % The item's ''object''
    /ObjectX         0 def     % and bounding rect:
    /ObjectY         0 def
    /ObjectWidth     0 def
    /ObjectHeight    0 def
    /ItemLabel       nullstring def   % The item's ''label''
    /LabelX          0 def     % and bounding rect:
    /LabelY          0 def
    /LabelWidth      0 def
    /LabelHeight     0 def
    /ItemBorder      2 def     % Extra space around the item
    /ObjectLoc       null def % Label-Object position
    /ItemGap         5 def     % Distance between object & label
    /ItemFrame       0 def     % Draw frame if not zero
    /ItemRadius      0 def     % Radius of frame
dictend
classbegin
% default variables
    /ItemLabelFont  Item /ItemFont get def
% class variable: over-ride of PaintItem
    /PaintItem   % – => –
% methods: over-ride new
    /new% label obj loc notify parent width height => instance
% utilities used to manipulate label-object pair
    /LabelSize   % – => width height
    /ShowLabel   % – => –
    /ShowObject  % – => –
    /EraseObject % – => –
    /AdjustItemSize % – => –
    /CalcObj&LabelXY % – => –
classend def
```

The label and object and their item-relative bounding rectangles are defined by **ItemLabel, LabelX, LabelY, LabelWidth, LabelHeight, ItemObject, ObjectX, ObjectY, ObjectWidth,** and **ObjectHeight**.

The **ItemLabel** and **ItemObject** are either a string, an icon keyword, or a procedure keyword. If it is a procedure, it takes a boolean as an argument: true causes it to draw itself; false causes it to return its width and height.

The **ItemLabelFont** is bound to the label, the **ItemFont** is bound to the object.

The item's layout metrics are defined by **ItemBorder, ObjectLoc, ItemGap, ItemFrame,** and **ItemRadius.**

ItemBorder is the space between the item bounding box and its label-object pair. **ObjectLoc** is the position of the object relative to the label. It may be any of **/Right, /Left, /Top, /Bottom. ItemGap** is the space between the label-object pair. **ItemFrame** is the size of the frame to draw around the item. It should be no greater than **ItemBorder. ItemRadius** is the curvature of the item's border. Zero represents a rectangular shape; a number between 0 and .5 represents a rounded rectangle whose radius is that fraction of the shortest edge; and any other number is used as the absolute curvature of the rounded rectangle.

The two overrides are the **/new** method and the **/PaintItem** procedure called by the **/paint** method. Note that **/new** adds **label, obj, loc,** and **notify** to the arguments of its superclass. These are bound to **ItemLabel, ItemObject, ObjectLoc,** and **NotifyUser,** respectively.

Class LabeledItem contains a few utilities that are used by its subclasses **LabelSize, ShowLabel, ShowObject, EraseObject, AdjustItemSize,** and **CalcObj&LabelXY.**

LabelSize returns the height and width of the label. **ShowLabel** and **ShowObject** paint the label and object in the item's canvas; **EraseObject** erases the object. **AdjustItemSize** and **CalcObj&LabelXY** are two layout utilities. **AdjustItemSize** is used to insure the item is large enough for its label-object pair while **CalcObj&LabelXY** is used to adjust the label-object pair's relative locations. Note that **CalcObj&LabelXY** adds the initial values of the label and object locations, which default to 0, to the calculated locations, thereby providing for slight adjustments by the programmer.

F.3 Subclasses of *LabeledItem*

This section presents several practical subclasses of **Class LabeledItem,** showing how they are used by client programs. There are further examples of item usage in the *itemdemo* program provided in the standard release. The implementation of these classes is included in the *item.ps* file that implements both **Class Item** and **Class LabeledItem**. Programmers wanting to implement their own items should look at these implementations; they are generally less than a page of PostScript.

The subclasses are:

- **ButtonItem**: provides a simple activation/confirmation item
- **CycleItem**: provides check boxes and choices
- **SliderItem**: provides a continuous range of values
- **TextItem**: provides a type-in area
- **MessageItem**: provides an output area
- **ArrayItem**: provides an array of choices

Things to notice:

- All items except button items have the following arguments:

  ```
  label object location notifyproc parentcanvas width height
  ```

  Buttons have no object; thus leave out **object** and **location**. Cycle and array items have multiple objects in an array.

- All four object locations are visible in the sample:

— The text and slider items use /**Right**

— The cycle item uses /**Left**

— The message item uses /**Top**

— The array item uses /**Bottom**

- The **width** and **height** parameters are hints only; the **AdjustItemSize** procedure will increase these if necessary. The minimal size will be enough to contain the label-object pair separated by *ItemGap* with a border of **ItemBorder**. The message item in the example below uses this by using a 0,0 size for the canvas but a large empty string as the initial value of the message.

- The values calculated by **CalcObj&LabelXY**

  ```
  LabelX LabelY ObjectX ObjectY
  ```

  can be adjusted by assigning initial values to any of them.

  ```
  /cycle (CycleItem) [/panel_check_off /panel_check_on]
      /Left /notify can 0 0 /new CycleItem send
      dup /LabelY -4 put 10 70 /move 3 index send def
  ```

- The label or object can generally be either a string, an icon name, or a procedure. The procedure takes a boolean which indicates whether to draw the object (true) or to return its width and height (false). The array object above shows both an icon and a drawing procedure (the box):

  ```
  /drawing {
      {ItemBorderColor setcolor 1 1 14 14 rectpath stroke}
      {16 16} ifelse
  } def
  ```

  Use of these items follows the general pattern:

- A canvas is created for containing the items. This is generally done by creating a window and getting its **ClientCanvas**. The window's **PaintClient** procedure should include a call to **paintitems**.
- A dictionary (or array) of items is created using the **/new** message to the item class of interest followed by a **/move** message to this instance.
- This collection of items is activated by calling **forkitems**. The items' notification procedures will be used to perform the activities the programmer desires.

Here is a minimal example of this style. We create a window with a button and a message item. The button's notify procedure simply prints ''Button Pressed!'' in the message item.

```
/win {
    /FrameLabel (Message Test) def
    /PaintClient {items paintitems} def
} makewindowfromuser def
/can win /ClientCanvas get def
/notify {(Button Pressed!) [ ] /printf
            items /message get send} def
```

```
/items 50 dict dup begin
    /message (Message:) (                               )
        /Right {} can 0 0 /new MessageItem send
        10 10 /move 3 index send def
    /button (Button) /notify can 0 0 /new ButtonItem send
        10 50 /move 3 index send def
end def
/p items forkitems def
```

F.4 *LabeledItem* Details

This section presents details on the use of the **LabeledItem** subclasses.

- **ButtonItem: ItemValue** is a boolean; true if pressed, false if not. Generally the **ItemValue** is never used; the **NotifyUser** is called to perform the button's activity. **ButtonItems** differ from the rest of the **LabeledItems** by not having an object and object location in the arguments to */new*.
- **CycleItem: ItemObject** is an array of objects. **ItemValue** is the index of the currently displayed object in that array. The cycle starts at zero and progresses one each ''push'' of the item. **NotifyUser** is called when the cycle changes value.
- **SliderItem**: The object is an array consisting of three integers: the minimum value, the maximum value, and the initial value for the slider. The **ItemValue** is the current value of the slider, and **NotifyUser** is called when the button is released. You can be notified continuously by overriding **ClientDrag**:

  ```
  /ClientDrag {/ClientDrag super send NotifyUser} def
  ```

- **TextItem**: **ItemValue** is the current string being displayed. The object is the initial string. **NotifyUser** is called whenever there is any change to **ItemValue**. Text items differ from the others in the way they use the mouse. **MouseDown** activates the text item if it is not yet active, and changes the caret location if it already active. The item is de-activated by activating another text item or by exiting the parent canvas of the text item. Keyboard motion is available in the text item using standard Emacs control sequences.

- **MessageItem**: The object in a message is its initial value. This need not be text! Message items have two additional methods: **/print** and **/printf**. **/print** takes a single argument, generally a string, and displays it as the item's new object. **/printf** has two arguments: a format string and an argument array. See the previous chapter for sample usage. **NotifyUser** is called whenever a new message is posted. It should generally be an empty procedure. **ItemValue** is the current message.
- **ArrayItem**: The object is an array of equal length arrays. The ''inner'' arrays are the subsequent rows. The sample array item was created by:

```
(ArrayItem) [
    [(One) (Two) /panel_text]
    [(Four) /drawing (Six)]
] /Top /notify can 0 0 /new ArrayItem send
```

ItemValue is an array of indices of the current selection and is initialized to [0 0]. **NotifyUser** is called from **ClientUp** if the **ItemValue** changed from its initial value.

Index

Section 3:

Programming the IRIS Window Manager

Contents

List of Tables

1. Introduction

Rather than implementing any specific user interface, the NeWS server provides only the most basic facilities necessary for window management and event handling. However, since NeWS can be extended via user-defined PostScript functions, you can create a sophisticated user interface that runs on top of the raw facilities provided by the NeWS server.

4Sight includes a set of PostScript files which define a high-level user interface within NeWS. Although these files do not represent a single entity, it makes sense to refer to them collectively as the IRIS Window Manager because the execution of these files by the NeWS server results in a user interface which would, under other window systems, be defined and controlled by monolithic window management programs.

The advantage of writing a window manager as a set of executable PostScript files instead of a monolithic, compiled program written in a language like C is that the interface becomes very easy to customize. Under 4Sight, it is a trivial matter to adjust the ''look and feel'' of the IRIS Window Manager's user interface to fit your preferences. For example, you can change the background or window frame colors, change the font used in menus or title bars, or even change the shape and dimesions of window borders simply by editing the definitions of a handful of PostScript functions. In the extreme case, you could completely rewrite the PostScript files so that the window manager's behavior is entirely different from the version of the window manager supplied with 4Sight.

Since the IRIS Window Manager's routines are written in PostScript, changing them is similar to writing PostScript client programs as described in section 2 of this manual, ''Programming in NeWS''.

Three major topics are covered in this part of the *4Sight Programmer's Guide*:

- using IRIS Window Manager extensions to NeWS
- using window, menu, and item packages
- customizing the desktop environment

There is also an appendix giving a brief description of each of the IRIS Window Manager PostScript flies.

1.1 What is a Window Manager?

A window manager is a user interface environment that controls multiple processes. Window managers include the following features:

- separate areas of a workstation screen to do parallel tasks simultaneously. These areas are generally called windows and can be arranged in different ways. For example, tiled windows are always adjacent to one another, and overlapping windows can be stacked, so some of the windows are not in full view. The IRIS Window Manager allows overlapping windows.
- mouse control to select items from various pop-up menus and other controls which activate different windows, change their position, or otherwise affect them
- configuration of windows, that is, the user can change the behavior of the input device and the color of the windows

The window manager environment has its own terminology that describes its various concepts and functions. This terminology is defined in the following section.

1.2 Window Manager Terminology

This section describes the terms and concepts associated with the IRIS Window Manager, which you will find throughout this document.

Each *window* provides an individual graphics context, e.g., a virtual graphics device, which is independent of all other windows. There are two types of windows: text windows, in which you edit text and enter operating system commands, and graphics windows, in which graphics programs are displayed.

A *graphics program* can draw graphics into one or more windows; it can also read input devices and create graphic images. An interactive program can make requests to change the position or appearance of a window. You can control the IRIS WIndow Manager using a procedural interface, i.e., certain routines from the Graphics Library or PostScript.

The window with the *input focus* is the window that currently receives input events, such as keystrokes, mouse position, and mouse button events, which go to the process.

The *cursor* is an object on the screen that follows the position of the mouse or other input device. A graphics program can change the current shape of the cursor, which is controlled by the process with the current input focus. The default cursor is an arrow pointing towards the upper-left.

Graphics programs and the window manager often use *pop-up menus*, which allow you to select (activate) an item on the menu. For example, the window menu provides items such as ‘Pop’ and ‘Move’, which allow you to bring a window to the top of a stack of overlapping windows and to move the window around the screen, respectively. You can create pop-up menus in your graphics programs using the pop-up menu routines.

The *frame* is the border of each window that always lets you access the ‘Window’ functions. Requesting a menu while the cursor is over the frame lets you change the shape or appearance of the window.

1.3 What Does the IRIS Window Manager Do?

The IRIS Window Manager lets you

- direct input focus to a specific window
- change which window is on top, when two or more are overlapping
- move windows
- reshape windows
- iconify windows
- delete windows

There is also a procedural interface to the window manager. Programmers can use Graphics Library routines to write a program that uses features of the window manager. When you program, you can use the GL interface routines to do the following:

- change the order in which the windows appear on the screen
- reshape windows
- move the input focus
- create pop-up menus

The GL interface routines are explained in Section 1 of this manual, ''Using the GL/DGL Interfaces''.

2. The Extended Input System

The input mechanisms described in Section 2 of this manual, Programming in NeWS, provided two things:

- a basic, default user interface, and
- a platform on which to build more sophisticated interfaces.

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

2.1 Building on NeWS Input Facilities

This chapter describes an *Extended Input System* (EIS) for NeWS. It is implemented entirely in PostScript, on top of the basic facilities provided by the primitives in the NeWS server. It aims to support a sophisticated user interface and to provide at least one such interface as an existence proof. It also is aimed at separating independent issues in the implementation of interfaces. For example, it should be possible to provide alternatives in each of the following categories without dependencies between categories and without requiring any change to client code:

- different input devices (1- and 3-button mice, or keyboards with different collections of function keys);
- alternative styles of input-focus, such as follow-cursor or click-to-type;

The EIS is sufficiently flexible that it should be possible to support a keyboard-only input system.

This chapter has several independent sections, corresponding to some of the modules of the EIS. It begins with a description of a particular user interface, implemented by the file *liteUI.ps*. It includes a description of the requirements and facilities for a client to handling keyboard input and selections in that world.

A good deal of the processing in the EIS is carried on in a single process called the ''global input handler.'' Some of it, however, must be done on a per-client basis; facilities are provided which are active in the client's lightweight process in the server. For example, recognizing events that indicate a change of input focus and distributing keystrokes to that focus are done in the global input handler. But recognizing user actions that indicate a selection is to be made must be done for each client, since some clients will not make selections at all, but will apply other interpretations to the same user actions.

2.2 The LiteUI Interface

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

- In an initialization phase, the client declares its interest in various classes of activity. These classes include simple and extended keyboard input, and selection processing. In response, the EIS sets up a number of interests (some in the global input handler, some in the client's own process), and records the client in some global structures.
- The client process enters its main loop, which includes an awaitevent. Some of the events it receives will be in response to interests expressed in the initialization calls it made. These events will generally be at a high semantic level; translating mouse events into selection actions is done inside EIS. The client will typically have more work to do with these events; for example, characters may be sent across the communication channel to be processed in the client's non-PostScript code. Some of the

processing will require calls back into EIS code; for example, a client will have to inform the system what selection it has made in response to selection events.

- Finally, when a client no longer requires various EIS facilities, it should revoke its interests, so that resources do not remain committed when it no longer needs them.

2.3 Keyboard Input

This section details the methods by which NeWS handles keyboard input. NeWS handles keyboard input in three basic contexts; simple ASCII characters, special function keys, and on-screen editing.

2.3.1 Simple ASCII Characters

Four procedures provide access to increasingly sophisticated levels of keyboard input. The most straightforward client merely wants to get characters from the keyboard. This is done by invoking **addkbdinterests** (passing the client canvas as an argument) and then sitting in a loop, doing an **awaitevent** and processing the returned event.

canvas **addkbdinterests** [events]

Declares the client as a candidate for the input focus. It also creates and expresses interest in the following three kinds of events, and returns an array of the three corresponding interest-events:

The first interest has **ascii_keymap** for its **Name**, and **/DownTransition** for its **Action**. (**ascii_keymap** is a dictionary provided by EIS for expressing interest in ASCII characters; it includes the translation from the user's keyboard to the ASCII character codes where that is necessary.) Events which match this interest will have ASCII characters in their **Names**, and **/DownTransition** in their actions. The client can choose to see up-events too, by storing **null** into the **Action** field of this interest.

The second interest has **/InsertValue** and a **null Action**. This will match events whose **Name** is the keyword **/InsertValue**, and whose **Action** is a string which is to be treated as though it had been typed by the user. Such events will be generated if some process is doing selection-pasting to this window, or if function-key strings have also been requested (see below).

The third interest has the array [**/AcceptFocus /RestoreFocus /DeSelect**] in its **Name**. Events matching this interest inform the client it has gotten or lost the input focus. (**/DeSelect** events referring to the focus will have an **Action** of **/InputFocus**.) These events are informational only; they do not affect the distribution of keyboard events. They are intended for clients which provide some feedback, such as a modified namestripe or a blinking caret, when they have the focus. Clients are always free to ignore them.

A process that continues to exist, but wants no more keyboard input, may revoke an interest in keyboard input by passing the array returned from **addkbdinterests**, along with the client canvas, to **revokekbdinterests**:

[events] canvas **revokekbdinterests** –

Undoes all the effects of **addkbdinterests**.

2.3.2 Function Keys

By default, clients do not receive any events associated with function keys. A client can choose to receive function-key events, either in the form of a keyword naming the key that went down, or as a string of the form `ESC [`*nnn*`z` (the ASCII-standard escape sequence for such keys).

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

canvas **addfunctionstringsinterest** event

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

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

canvas **addfunctionnamesinterest** event

creates an interest in the function keys, expresses interest in it, and returns that event. As a result, when a function key is pressed, **awaitevent** returns an event whose **Name** is a keyword like **/FunctionF7**. By default, both up and down transitions on the keys are noted; the client may change this by storing **/DownTransition** (or **/UpTransition**, if that is what is desired) into the **Action** field of the returned interest. **addkbdinterests** must also have been called for this procedure to have any effect.

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

2.3.3 Keyboard Editing and Cursor Control

If the client is passing characters through to a shell or some similar process that will do its own translations on them, they should be passed unmodified. But if the client is dealing with text directly, it should provide the editing and caret-motion facilities defined in the user's global profile. To assist in this, the client may ask for incoming events to be checked for a match against those keyboard actions, and converted to uniform editing-events if they do. This is done by passing the client canvas to **addeditkeysinterest**.

canvas **addeditkeysinterest** event creates an interest in the key combinations that are defined for global editing and caret motion, expresses interest in it, and returns that event. As a result, the client sees events with a **Name** from the set:

{Edit, Move} {Back, Fwd} {Char, Word, Field, Line, Column}

For example, here are the key combinations for **EditBack***:

EditBackChar	**null**
Delete the character before the caret	
EditBackWord	**null**
Delete the word before the caret	
EditBackField	**null**
Move the caret back to the end of the preceding field if any exists, deleting its contents or selecting them in pending-delete mode	
EditBackLine	**null**
Delete from the caret back to the beginning of the current line	
EditBackColumn	**null**
Delete all characters between the caret and the nearest boundary in the line above; if the previous line ends to the left of the caret, delete back through the preceding end-of-line	

Substituting **Fwd** for **Back** indicates the deletion or motion (see the next paragraph) extends *after* rather than before the caret. **EditFwdLine** deletes through the next end-of-line.

Substituting **Move** for **Edit** indicates the caret is moved to the far end of the span that would be deleted by an **Edit**, but the characters are not deleted.

Again, no separate procedure is provided to revoke this interest, since the **revokeinterest** primitive does exactly what is needed.

2.4 Selection Overview and Data Structures

Clients that will make selections and pass information about them to other processes declare this interest via **addselectioninterests**. Thereafter, EIS code will process user inputs according to the current selection policy. Occasionally, it will pass a higher-level event through to the client, when some client action is required in response. The exact interface by which a user indicates a selection is not the client's responsibility; the client must simply be prepared to handle higher-level events. Clients will also occasionally see events with a **Name** of **/Ignore**; these are events which were delivered to the client process, but handled entirely by EIS code before the event was made available to the client. The **/Ignore** event is left behind in this situation so that client code can depend on an event being on the stack when it gets control after **awaitevent** returns.

2.4.1 Selection Data Structures

There is no separate ''selection service'' in EIS; some selection processing takes place in the global input handler, and the rest in client processes. There is a global repository of data about selections, however, and there are some standard formats for the information stored in that repository and communicated between selection clients.

A selection is named by its *rank*; in *liteUI*, the ranks are **/PrimarySelection, /SecondarySelection**, and **/ShelveSelection** (There is nothing to prevent clients from using other ranks, with names they define themselves. Strictly speaking a rank is simply a key in the **Selections** dictionary).

For each rank, there is a dictionary containing the information known to the system about that selection. Such a dictionary will be called a *selection-dict* henceforth. It will have at least the three keys defined in Table 2-1.

Key	Type	Semantics
/SelectionHolder	*process*	the process that made the selection
/Canvas	*canvas*	the canvas in which the selection was made
/SelectionResponder	*null / process*	the process that answers requests concerning this selection

Table 2-1. Selection-dict Keys

If **/SelectionResponder** is defined to **null**,
there will be other keys defined in the dictionary,
setting out all available information about that selection.
A few such keys have been defined because they are expected to be generally useful.
These are listed in the table below.
Others may be provided by clients as convenient –
there is no limit on what consenting clients may say to each other about a selection.

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

Table 2-2. System-defined Selection Attributes

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

a request-dict will contain the keys defined by the requester, plus the two keys in the Table 2-3.

Key	Type	Semantics
/Rank	*rank*	the rank selection this request concerns
/SelectionRequester	*process*	the process which is sending the request

Table 2-3. Request-dict Entries

The use of these various structures is detailed under the relevant event descriptions below.

2.4.2 Selections: Library Procedures

This section lists the library procedures provided for clients to deal with selections.

canvas **addselectioninterests** [events]

Creates and expresses interest in two classes of events, returning an array of the two interests.

The first interest matches events with names in the following list:

/InsertValue
/SetSelectionAt
/ExtendSelectionTo
/DeSelect
/ShelveSelection
/SelectionRequest

The response required from the client to each of these events is detailed below. (Some clients may safely omit handlers for the last two; see the detailed description).

The second interest matches events which are uninteresting to the client. It arranges for EIS processing to be done by library code before the client ever sees the event.

rank **clearselection** –

Sets the indicated selection to null; this allows a selection holder to indicate the selection no longer exists.

request-dict rank **selectionrequest** request-dict

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

event **selectionresponse** –

Used by a selection holder to return a response to a selection request. The argument should be a **/SelectionRequest** event that has been processed by the holder. (**/SelectionRequest** events are described below. The event will be transformed into a **/SelectionResponse** event and returned to the requester.

selection-dict rank **setselection** –

Used by a process to declare itself the holder of a selection. *Selection-dict* is a dictionary containing either a definition of **/SelectionResponder**, or of keys which provide data about the selection itself, as described above in Table *Selection-Dict Keys*. *Rank* indicates which selection is being set. If another process currently holds that selection, it will be told to deselect.

rank **getselection** selection-dict

Retrieves the information currently known to the system about the indicated selection. This procedure is likely to be more useful to the implementor of a package like *liteUI* than to window clients.

2.4.3 Selection Events

As mentioned above, clients may expect to receive six different kinds of events concerning the selection. Of these, the **InsertValue** event has already been described under *Keyboard Input*; its usage in the selection context is exactly the same as for function strings. The remaining five events and the appropriate responses to them are presented below.

Each event is described in the following format:

/EventName

Semantics: *short description of the event*

Action: *description of the contents of the event's* **Action** *field*

Response: *description of what the client should do when it receives such an event*

/SetSelectionAt

Semantics: Informs the client the user has just made a selection in its canvas.

Action:

```
dict [          Rank             /PrimarySelection |
                                 /SecondarySelection
                X                number
                Y                number
                PendingDelete    true | false
                Preview          true | false
                Size             number
]
```

LiteUI provides constant values for three fields: **PendingDelete** = **false**, **Preview** = **false**, and **Size** = 1.

Response: Make a selection of the indicated **Rank** with the following parameters:

Key	Value
X and **Y**	indicates a position (it will be in the current canvas' coordinate system).
Size	indicates the unit to be selected; for example, in text: **0** means a null selection at the nearest character boundary, **1** corresponds to a character, and larger values indicate larger units (words, lines, etc.) whose definition is at the discretion of the client
PendingDelete	indicates whether that mode should be used (if supported by the client)
Preview	indicates whether the selection is only for feedback to the user; a selection shouldn't actually be set until a selection event is received with **Preview false**

Table 2-4. SetSelectionAt Parameters

In client PostScript code, some private processing will generally be required. For instance, the given position will have to be resolved to a character in a text window, and appropriate feedback displayed on the screen. Then the client should build a selection-dict describing the selection just made, and pass it to **setselection,** along with the rank it received in the /**SetSelectionAt** event:

```
selection-dict rank setselection
```

selection-dict should contain either a non-null definition of /**SelectionResponder**, or it should define keys which actually provide information about the selection (/**ContentsAscii** at a minimum). In the former case, the holder is following a *communication-model* of selection, and must be prepared to receive and respond to /**SelectionRequest** events as long as it holds the selection. In the latter case, the holder is following a *buffer-model* of selection; requests will be answered automatically by the global input handler.

selection-dict will have keys added to it, so it should be created with room for at least five more entries beyond those defined by the client.

/ExtendSelectionTo

Semantics: Informs the client the user has just adjusted the bounds of a selection in its canvas.

Action:

```
dict [          Rank             /PrimarySelection |
                                 /SecondarySelection
                X                number
                Y                number
                PendingDelete    true | false
                Preview          true | false
                Size             number
]
```

Response: The dictionary in the **Action** field is the same as the **Action** of a **/SetSelectionAt** event, and the client response is very much the same. The distinction is that this event indicates a modification of an existing selection, where **/SetSelectionAt** indicates a new one.

The client should adjust the nearest end of the current selection of the indicated **Rank** to include the indicated position. If **Size** indicates growth, extend both ends as necessary to get them at a boundary of the indicated size. (For example, if **Size** has changed from 1 to 2, a text window might grow both ends of the selection to ensure that they fall at word boundaries.) Adjust the **PendingDelete** mode or ignore it as the window is editable or not.

If there was no selection of the indicated rank, pretend there was an empty one at the indicated position.

In client PostScript code, after doing any private processing required, processing is exactly the same as for **/SetSelectionAt**.

/DeSelect

Semantics: Informs the client that it no longer holds the indicated selection.

Action: *rank*

Response: Undo a selection of the given rank in this window. *Do not* call **clearselection**; the global selection information has already been updated.

/ShelveSelection

Semantics: Tells the client to set the shelf selection to be the same as a selection which the client currently holds.

Action: *rank*

Response: Buffer-model clients (those that did not define **/SelectionResponder** when they set the selection) will not receive **ShelveSelection** events; the service will copy their selection to the shelf for them. Others should set the **ShelveSelection** to be the same as the selection whose *rank* is in **Action**, using **setselection** as above.

/SelectionRequest

Semantics: The client is requested to provide information about a selection it holds.

Action: request-dict

Response: Buffer-model clients (those that did not define **/SelectionResponder** when they set the selection) will not receive **SelectionRequest** events; the service will answer the request for them.

The client should enumerate the request-dict, responding to the various requests by defining their values (as for **/ContentsAscii**), or performing the requested operation (as for **/ReplaceContents**, whose value will be the replacement value). The resultant dict should be left as the **Action** of the event, which should then be passed as the argument to the procedure **selectionresponse**.

There is no restriction on what requests may be contained in a selection request; this is left to negotiation between the requester and the selection holder. A holder may reject any request, by defining its value to be **/UnknownRequest**.

It may be noted that there is no mechanism described here for getting a selection's contents from someplace else. In *liteUI*, user actions that precipitate such a transfer are recognized and processed in the global input handler, which then performs the selection request, and sends an **/InsertValue** event to the receiving process. The selection library procedures described above provide an interface for instigating such transfers independent of user actions.

2.5 Input Focus

The input focus (where standard keyboard events are directed) is maintained by the global input-handler process, according to the current focus policy. A client becomes eligible to be the input focus by calling **addkbdinterests**. At some later time, some user action will indicate that the client should become the focus. The client will receive an event indicating this has happened (its **Name** will be **/AcceptFocus** or **/RestoreFocus**, and its **Action** will be **null**). Thereafter, the client will receive events whose **Names** are ASCII character codes.

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

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

canvas process **setinputfocus** –

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

– **currentinputfocus** [canvas, process]

The current input focus is returned by **currentinputfocus**. If there is no current focus, **null** is returned.

process **hasfocus** boolean

Returns **true** or **false** as the indicated process is or is not currently the input focus.

keyword **setfocusmode** –

The global focus policy is reset to the policy named by the argument. Currently-supported focus policies are identified by:

/MaxFocus

a window will receive the focus when the mouse enters its subtree, and lose it when the mouse exits. If the mouse crosses window boundaries while any key or button is down, a focus change is delayed until all keys or buttons are up, and then reflects the current situation.

3. Window and Menu Packages

This chapter presents two class-based packages: one for pop-up menus, one for windows. We first discuss the notion of a package in general, then give the client interface specification for both packages followed by a detailed sample program. We then show how packages are installed into the system during the initialization of the server. The default user interface is then presented showing how these defaults can be changed.

To emphasize that these are ''lightweight'' implementations, these classes are called **LiteMenu** and **LiteWindow**. Their implementation files are *litemenu.ps* and *litewin.ps* in */usr/NeWS/lib/NeWS*.

3.1 Package Style

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

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

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

Many classes have several subclasses that can be used in place of the class itself. Thus one might have three subclasses of class '**Foo**' that can be used anywhere '**Foo**' can. For example, suppose we have three interface styles: '**MexFoo**,' '**MacFoo**,' and '**NewsFoo**.'

To allow the user to specify a preference, we use a variable, '**DefaultFoo**,' that has assigned to it the current preference:

```
/DefaultFoo MexFoo store
```

Note the use of **store** rather than **def**; '**DefaultFoo**' already exists and you want to change its value, not possibly create a new one on the stack. The initial values for **DefaultMenu** and **DefaultWindow** are **SGIGoodMenu** and **SGIWindow**, respectively.

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

3.1.1 Description Format

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

arg1 ... argN `/samplemethod Foo send`

the interface description will look like:

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

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

3.2 Menu Methods

Menus associate a key, generally a string, with an action to be performed when that key is selected by the user. If the menu action is another menu, nested displaying of that menu is performed. The default user interface style uses a pull-right nesting.

NeWS favors greater use of multiple lightweight processes. While the menu is being tracked, other computing may also be performed. In particular, NeWS menus do not freeze or lock the screen. This is considered bad manners unless there is good reason for such locking behavior.

The following methods are used with menus.

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

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

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

creates '**MyMenu,**' which displays the strings

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

and associates the actions

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

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

A key may also consist of an array of strings and/or icons. The following code displays a pretty arrow instead of the "=>" used in the menu from the previous page.

```
[(Other) pullRightIcon] MyOtherMenu
```

The code below draws a one-pixel wide horizontal line between the second and third menu entry of the menu on the previous page.

```
[(Key 2) 1 /MenuLine] {menuproc2}
```

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

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

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

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

sets the point size to be the integer value of the selected key. Note that we did not need to use **/currentkey self send** in this situation. The action procedure is called within the scope of the menu instance, and thus does not need to re-establish the menu's context.

– /popup –

Pop-up the menu and track user actions; call the associated action if a key is selected. This is generally called from an event manager as shown below. This call is often not required because of the **ClientMenu** feature of windows discussed later.

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

See Section 2, Chapter 5 of this manual, ''New PostScript Operators'', for descriptions of **eventmgrinterest** and **forkeventmgr**.

/popup is a direct replacement for **/show**.

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

key **/searchkey** index bool

Searches for a given key's position in the menu and returns the index of the key if found (along with a boolean of true) or a boolean of false if the key is not found.

action **/searchaction** index bool

Searches for the given action's position in the menu, returning its location and a boolean of true if found, or a boolean of false otherwise.

– **/currentkey** key

Returns the selected key when called within a menu action procedure.

– **/currentindex** index

Returns the index of the selected key when called within a menu action procedure. The indices begin at zero.

index key action **/insertitem** –

Insert a new key/action pair into an existing menu at the given index. Thus:

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

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

index **/deleteItem** –

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

```
0 /deleteitem MyMenu send
```

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

index key action **/changeitem** –

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

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

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

string **/settitle** –

Set the title bar of the menu to *string*. Thus:

```
(Foo) /settitle MyMenu send
```

would set the title bar of the menu MyMenu to read ''Foo''. This method works only for menus of class SGIMenu (the default menu for 4Sight).

3.3 Window Methods

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

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

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

canvas **/new** window

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

First, the standard creation of a window:

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

Then, the modification of the window:

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

Table 3-1 contains the instance variables that are commonly modified, along with their initial values; there are other instance variables available, but the interface to them is prone to change.

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

Table 3-1. LiteWindow Instance Variables

– **/destroy** –

Destroy the window and its entire process group. (UI)

x y width height **/reshape** –

Reshape the window to have the given bounding box.

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

– **/reshapefromuser** –

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

– **/map** –

Make the window/icon visible. Fork the window's event manager if that has not already been done. **/map** is initially called by the client, but is then handled by the window's user interface. (UI)

– **/unmap** –

Make the window/icon invisible. (UI)

– /flipiconic –

Alternate between opened (window) and closed (iconic) state. (UI)

x y /move –

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

– /paint –

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

– /paintclient –

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

– /paintframe –

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

– /painticon –

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

– /totop –

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

– /tobottom –

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

3.4 Example Program: lines

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

```
#! /usr/NeWS/bin/psh

%  Draw a window with a bunch of lines.
%  The icon uses the same drawing procedure as the client.

/fillcanvaswithlines {              % linesperside => –
    gsave
    1 fillcanvas % Paint the background
    0 setgray                       % Set the default color black
    clippath pathbbox scale pop pop                 % Set scaling to be 0 to 1
    0 1 3 -1 roll div 1 {                           % 0 delta 1 {..} for
        ColorDisplay? {dup 1 1 sethsbcolor} if
        0 0 moveto 1 1 index lineto stroke          % Draw line to top
        0 0 moveto 1 lineto stroke                  % Draw line to side
        pause                       % Let others run
    } for
    grestore
} def

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

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

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

main
```

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

The '**fillcanvaswithlines**' procedure takes as an argument the number of lines per side to draw. An important point to notice here is the use of **pause** in the **for** loop. This allows the lightweight process mechanism to optimize interactive behavior. Be sure to use **pause** in any part of your program that will potentially take a long time. In this case, if the user has selected 1000 lines, the drawing could last several seconds. However, there is a cost in using **pause**; a more efficient version of the **lines** example would pause every 10-20 sets of vectors.

The procedure '**main**' initializes the '**linesperside**' parameter, defines a menu action procedure, '**setlinesfrommenu**', and initializes the program's window.

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

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

instead. Finally, '**setlinesfrommenu**' causes the client canvas to be repainted by sending **/paintclient** to '**win**.' We do this rather than simply calling '**fillcanvaswithlines**' directly in order to inherit window manager side effects (mainly the use of forking the client paint procedure).

The window initialization consists of creating a default window, installing our modifications, setting its shape from user input, and finally making it visible. The modifications we install set the frame label to the string (''Lines''), set the client repaint procedure to call '**fillcanvaswithlines**' with the current value of '**linesperside**', set the icon to draw itself using '**fillcanvaswithlines**,' and finally set the client canvas' menu to be one that resets the '**linesperside**' parameter. Note that we map the window as our last step. This is intentional — mapping should be deferred until all setup is performed.

3.5 Default User Interface

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

Button Name	Initial Value
PointButton	LeftMouseButton
AdjustButton	MiddleMouseButton
MenuButton	RightMouseButton

Table 3-2. Window User Interface Button Usage

You may want to redefine these according to taste.

3.5.1 PointButton

PointButton operates as follows:

- In a window frame's ''stow'' region (top left): causes the window to become a window icon.
- In a window frame's ''close'' region (top right): closes the window.
- In a window frame's ''resize'' regions (the bracket at the four corners of the window frame): lets the user resize the window from those corners.
- Elsewhere in the frame: brings the window to the top of the window pile if clicked, moves the window without popping if dragged.

- Anywhere in an icon: opens the window when clicked.

3.5.2 AdjustButton

AdjustButton operates as follows:

- Anywhere in a window frame (except the stow, close or resize regions): brings the window to the top of the window pile if clicked, moves the window without popping when dragged.
- Anywhere in an icon: moves the icon when dragged.

3.5.3 MenuButton

MenuButton operates anywhere in a window frame (except the stow, close, and resize regions) or icon: pops up the window or icon menu, respectively.

3.5.4 Modifying the User Interface

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

% Swap the Adjust and Menu mouse buttons.

```
/PointButton     LeftMouseButton def
/AdjustButton    RightMouseButton def
/MenuButton      MiddleMouseButton def
```

% Change the font in menus, and box the selected menu item instead of highlighting it.

```
DefaultMenu begin
    /MenuFont /Times-Italic findfont 14 scalefont def
    /StrokeSelection true def
end
```

% Change the font for frame headers.

```
DefaultWindow begin
    /FrameFont /Times-Bold findfont 12 scalefont def
end
```

3.6 A Scrollbar Implementation

ScrollingWindow is a subclass of **LiteWindow**. It is defined in */usr/NeWS/lib/NeWS/litewin.ps*. There are two simple scrollbars in the frame margin. Two classes, **ScrollbarItem** and **SimpleScrollbar** provide the basis for the implementation. The former is an abstract superclass that reflects the structure of scrollbars but does not implement one entirely. **SimpleScrollbar** implements a simple, one button scrollbar. The two scrollbars in **ScrollingWindow** are initialized to return values between 0 and 1. When viewing a typical document, this corresponds to a postion where 0 indicates the beginning, and 1 indicates the end.

3.7 Polymorphic Menu Keys

Menu keys may be strings, icon names, procedures, or class instances. The string and icon names simply display the corresponding text or object. Menu keys may also be wrapped in an array. This allows font and color changes, and slight adjustments in the x,y location of the key relative to its default position. It also allows you to pass additional arguments to a procedure or class instance. Thus:

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

is a key that displays 'MyIcon' in red with a slight x,y offset.

3.8 Menu Layout Variations

The **LitePullRightMenu** subclass of class LiteMenu allows general item layout. The layout is controlled by the class variable **/LayoutStyle**, which may be set to **/Vertical**, **/Horizontal**, or an array of rows and columns. The default style is **/Vertical**. See *$NEWSHOME/demo/colornames* for an example of a program using advanced menu layouts.

4. Customizing the IRIS Window Manager Environment

One of the most important features of the IRIS Window Manager is the ease with which you can modify it to suit your individual tastes. In addition to the extensible facilities provided by NeWS, the IRIS Window Manager contains a number of special extensions that make it particularly easy to put add new tools and functions, or tailor existing ones to fit your needs.

This chapter describes six ways you can modify the IRIS windowing environment to suit your needs and tastes.

- placing windows automatically
- working with toolchests
- placing window icons and toolchests
- customizing menu selection
- creating window icons
- redefining Window Manager PostScript functions

4.1 Placing Windows Automatically

When you log in to your IRIS, you may want some additional tools besides the toolchests and the console window to be at your immediate disposal without the need for typing any commands or selecting any menus. You would probably also want the Window Manager to automatically place the tools in the same position on the screen every time you log in. For example, you might want the Window Manager to put a clock in the lower right corner of the screen, and a calendar in the upper right corner.

4.1.1 Using *prefposition* and *preforigin*

Two functions in the Window Manager, **prefposition** and **preforigin,** position the windows of programs that you specify. You call these functions from within your *user.ps* file. **prefposition** allows you to specify the size and position of a window. **preforigin** allows you to specify the postion of a window, but not its size.

Note: **prefposition** is functionally identical to **makepreference** in 4Sight versions 1.0 through 1.3. **makepreference** is supported in 4Sight version 1.4 for purposes of compatibility, but support in subsequent releases is not guaranteed.

(name) x y width height **prefposition** –

name is the the name of the program as assigned by `winopen` in Graphics Library programs, or by **IconLabel** in NeWS programs. *name* is passed as a string, hence the enclosing parentheses. *x* and *y* are the horizontal and vertical position of the lower left corner of the window. *width* and *height* are the horizontal and vertical dimensions of the window. The *width* and *height* of Graphics Library programs are subject to constraints set by the `winconstraints` subroutine within the program. For Graphics Library programs, these four measurements are made in pixels; for NeWS programs, the measurements are made in screen points.

Note: The IRIS-4D screen measures 1280 pixels wide by 1024 pixels high. In the NeWS coordinate system it measures 960 points wide by 768 points high.

Some Graphics Library programs don't allow you to specify their size. You can use **preforigin** to specify the position of these windows on the screen. **preforigin** does not work with programs that do not specify their own size, including all NeWS programs. The syntax for **preforigin** is described below.

(name) x y **preforigin** –

name is the the name of the program as assigned by `winopen.` *name* is passed as a string, hence the enclosing parentheses. *x* and *y* are the horizontal and vertical position of the lower left corner of the window, measured in pixels.

Note: **preforigin** does not work with NeWS programs.

To place a square clock in the lower right corner and a perpetual calendar in the upper right corner, edit your *user.ps* file to look like the one below. If you do not already have a *user.ps* file in your home directory, copy the default one from */usr/NeWS/lib* to your home directory.

```
%
% User customization file.
%

/RestartActions [
    { (demochest) forkunix }
    { (clock) forkunix }
    { (ical) forkunix }
] def

(clock) 1150 20 110 110 prefposition
(cal) 1000 800 preforigin
```

The next time you log in to the system, the Window Manager will place the clock and calendar automatically.

4.1.2 Stacking Windows

If you have used **prefposition** or **preforigin** for a particular program, and invoke several versions of the program, each subsequent invocation is offset (by default) diagonally downwards and to the right, so that the title bar of each window is visible. Subsequent invocations of a program that has been assigned a **prefposition** or **preforigin** are placed using the following parameters:

```
UserProfile /prefStackSize  % integer number of windows in a
                            % stack
UserProfile /prefStacks     % integer number of stacks before
                            % windows wrap around
UserProfile /prefDelta      % array [ XDelta YDelta /Filling ]
```

The following variants of **preforigin** and **prefposition** can be used to customize the stacking parameters:

```
(name) X Y [ XDelta YDelta /Filling ] preforigin
(name) X Y W H [ XDelta YDelta /Filling ] prefposition
```

XDelta and **YDelta** set the horizontal and vertical offset from the previously invoked program. **/Filling** causes empty places in the stack to be filled in as new versions of the program are invoked.

The default values for the stacking parameters are shown below:

```
prefStackSize    4
prefStacks       2
prefDelta        [ 8 -30 /Filling ]
```

4.1.3 Using *prefalias*

You can set groups of programs with different names to stack together by using the **prefalias** function.

(name) (groupname) **prefalias** –

> *name* is the name of the program you are aliasing. *groupname* is the name of the collective alias to which are assigning various program names.

The following example illustrates the use of **prefalias** in your *user.ps* file.

```
(Shell-Group) 100 200 preforigin
(wsh) (Shell-Group) prefalias
(xterm) (Shell-Group) prefalias
```

Any windows named "wsh" or "xterm" will appear in a stack starting at 100,200.

4.1.4 Removing Window Placement Preferences

Once 4Sight has loaded your *user.ps* file containing a call to **prefposition** or **preforigin** for a particular program, each invocation of that program will appear in the same designated spot. If you want to remove the placement preference for a program, you can use the the Window Manager function **removepreference** to do so. The easiest was to remove a preference with **removepreference** is to use *psh*. See Section 2, Chapter 2 of this manual, ''Interacting with NeWS'' for information on using *psh*.

(name) **removepreference** –

> *name* is the name of the program as assigned by `winopen` in Graphics Library programs, or by **IconLabel** in NeWS programs. *name* is passed as a string, hence the enclosing parentheses.

The following session of *psh* removes the placement preference on the square clock that was set in the example *user.ps* file on the previous page.

```
% psh
executive
Welcome to 4Sight Version 1.4
(clock) removepreferences
bye
psh: NeWS server disconnected
%
```

4.2 Working With Toolchests

The IRIS Window Manager provides toolchests as an easy method of adding functionality to the IRIS desktop. A toolchests is, in essence, a small permanent window that is linked to a menu of items which either invoke windowing programs or execute PostScript code that affects the windowing environment.

4.2.1 Customizing the *Tools* Toolchest

The best way to learn how to create your own toolchest is by example. Below is the definition of the Tools toolchest. It can be found online in the file */usr/sbin/toolchest.* Copy it into your home directory.

```
#! /usr/NeWS/bin/psh

%
% Run program in demo dir.
%

/RunDemoProg {
    (NEWSHOME) getenv (/demo/) append exch append forkunix
} def
```

```
%  Clock submenu
%

/clockmenu [
    (Square Clock) { (clock) forkunix }
    (Round Clock) { (roundclock -s) RunDemoProg }
    (Modern Clock) { (roundclock -f) RunDemoProg }
] /new DefaultMenu send def
(Clocks) /settitle clockmenu send

%
%  Tool Chest
%

/Tools [
    [(Shell) 1 /MenuLine]   { (wsh -f Screen-Bold.15) forkunix }
    (Showmap)               { (/usr/sbin/showmap) forkunix }
    (Makemap)               { (/usr/sbin/makemap) forkunix }
    [(Clocks) pullRightIcon] clockmenu
] (Tools) /new ToolChest send def
```

As you can see, the definition of a toolchest is comprised primarily of menu lists, which, as you learned in Chapter 3, consist of key and action pairs.

Let's say you want to add the program *cedit* to the Tools menu, right after the entry for *showmap*. To do this, add the following line after the line containing 'Showmap'.

```
(Cedit)          { (/usr/sbin/cedit) forkunix }
```

Now, log out from the Window Manager, and log back in again. Look at the Tools menu; it now contains 'Cedit' as a menu entry. Note that the Window Manager ran the version of *toolchest* that you modified in your home directory; if it did not, make sure your path is set to look in your current directory first (edit your *.login* file and then type 'rehash' to reset your path).

4.2.2 Creating a New Toolchest

Now that you have modified a toolchest, you may want to try creatinig a new one. Perhaps, for example, you would like to move the clocks in the Tools toolchest to their own toolchest.

Remove all lines concerning clocks from your copy of *toolchest*. Then create a new file in your home directory called *clockchest*.

Place the following toolchest definitions in *clockchest*:

```
#! /usr/NeWS/bin/psh

%
% Run program in demo dir.
%
/RunDemoProg {
    (NEWSHOME) getenv (/demo/) append exch append forkunix
} def

%
%
%  Clock Chest
%

/Clocks [
    (Square Clock) { (clock) forkunix }
    (Round Clock) { (roundclock -s) RunDemoProg }
    (Modern Clock) { (roundclock -f) RunDemoProg }
] (Clocks) /new ToolChest send def
```

Make *clockchest* executable using the *chmod* command:

```
chmod +x clockchest
```

Edit your *user.ps* file. Directly below the line containing **RestartActions** add the following line:

```
                    { (clockchest) forkunix }
```

Now log out using the Window Manager menu, and then log back in again. You now have a new Clocks toolchest.

4.2.3 Example Toolchest: *Hosts*

This example toolchest is very useful if you have accounts on various other computers that you access from your IRIS via TCP/IP. This toolchest provides a menu of host machines. When you select one, you get a new *wsh* window with the remote machine's name as a title. It connects you to the remote machine via the *rlogin* command.

```
#! /usr/NeWS/bin/psh

/forkwsh { % args => -
    (wsh -r1000 -f Screen-Bold.15 -C 6,15,3,2) exch append
    forkunix
} def

/RemoteHostChest [
    (machine1) (machine2) (machine3) (machine4) (machine5)

]
[ {
    currentkey dup (-s40x80 -n ) exch append
    ( -c rlogin ) 3 -1 roll append append
    forkwsh
} ]
(Hosts) /new ToolChest send def
```

Replace 'machine' entries in the example with names of the remote hosts on which you have account names that are the same as your account name on the IRIS. If your account name is different, you can still include it as a menu entry by replacing the 'machine' entries in the example with entries of the following form:

(*remote_host* -l *remote_login*)

remote_host is the name of the remote machine; *remote_login* is the name of your login account on the remote machine.

Place the code and entries above into a file in your home directory named *hostchest*. Make the file executable with the *chmod* command, and add the following line to your *user.ps* file, directly below the line containing **RestartActions.**

```
                    { (hostchest) forkunix }
```

Log out using the Window Manager menu. When you next log in, you will see the Hosts toolchest.

4.3 Placing Icons and Toolchests

The position of icons (stowed windows) and toolchests on the IRIS screen is governed by two *tilers*, one controlling toolchest placement, and one controlling icon placement. You can specify your own paramenters to the tilers that control exactly where and when icons and toolchest are positioned on the screen. You make these specifications from within your *user.ps* file.

4.3.1 Tiling

Toolchests are placed using the **ToolTiler**. Icons are placed using the **IconTiler**. These tilers function within a specified area of the screen. You can change the default area for tiling in your *user.ps* file.

The following lines change the tiling area for icons and toolchests if you place them in your *user.ps* file:

```
x y width height /newarea IconTiler send
x y width height /newarea ToolTiler send
```

x and *y* is the origin of the tiling area on the screen, and *width* and *height* are its dimensions in NeWS points.

You can also control how each tiler lays out its icons or toolchests within the tiling area. You can control which corner of the tiling area the icons or toolchests are tiled from, and whether the tiler places them horizontally or vertically.

The following lines change the corner of origin for tiled icons and toolchests if you place them in your *user.ps* file:

```
corner /corner IconTiler send
corner /corner ToolTiler send
```

corner is one of the following:

```
/upperleft
/upperright
/lowerleft
/lowerright
```

The following lines change the direction in which icons and toolchests are tiled from the corner specified in the lines above:

```
*direction* /direction IconTiler send
*direction* /direction ToolTiler send
```

where *direction* is one of the following:

```
/horizontal
/vertical
```

The default layouts for icons and toolchests are as follows:

```
/upperleft /corner IconTiler send
/vertical /direction IconTiler send

/upperright /corner ToolTiler send
/vertical /direction ToolTiler send
```

4.3.2 Tidying Icons Automatically

By default, icons are tidied automatically whenever they are stowed; they are immediately positioned by the tiler. You can change this with the following line in your *user.ps* file:

```
UserProfile /TidyState *state* put
```

where *state* may be one of the following:

```
/Always
/First
/Never
```

/Always is the default; icons are always tidied when stowed.

/First tidies the icon only when it is first stowed.

/Never means the icon is not automatically tidied.

4.4 Customizing Menu Selection

By default, when a menu is popped up, the mouse cursor is over the title bar of the menu. You can automatically place the cursor over the first item in any menu by adding the following line to your *user.ps* file:

```
UserProfile /MenuHome /overFirst put
```

The default value is:

```
UserProfile /MenuHome /overTitle put
```

4.5 Creating Window Icons

When an active window is stowed by the user, an RGB image file may be used to paint the canvas of the stowed window icon. A window icon file must contain the suffix *.icon.*

Window icons are assigned to stowed windows by matching the name that appears in the program's call to the Graphics Library subroutine, `winopen`. Thus, an icon for the GL program *cedit* would have this name:

```
cedit.icon
```

4.5.1 The Window Icon Search Path

A directory of default window icons exists in *$NEWSHOME/icons*. You may add or customize window icons by placing them in *$HOME/.4sight/icons*. To find the appropriate window icon for a stowed window, 4Sight first searches *$HOME/.4sight/icons* for a name match. If it is unsuccessful, it seaches the default window icon directory (*$NEWSHOME/icons*). If this is unsuccessful, it uses the prededined icon *$NEWSHOME/icons/default.icon*. If this icon is missing for some reason, 4Sight draws the icon without an image.

4.5.2 Getting an Image

A window icon can be created from any image that can be displayed on the IRIS screen (provided that the tools described below are accessible when the image is displayed).

The following passage describes one possible way to create an icon image file. First, display the image you wish to use with the image tools *ipaste* or *showci*, or simply open a window containing a program from which you want to take an image (make sure that the image is still). Then invoke the image tool *icut* from the command line. *icut* takes a filename as an argument; the image cut from the screen is written to that file.

```
icut foo
```

To use *icut*, place the small rectagular *icut* window away from the image you wish to cut. Then place the mouse cursor inside the icut window and hold down the <shift> key. While holding down the key, move the mouse cursor to the upper left corner of the area you wish to cut. Hold down the left mouse button while you move to the lower right corner of the area you wish to cut (the area is not shown on the screen), and when you are ready, let go of the left button. The image is then written to the file.

Note: The image you cut must be scaled to fit the stowed window canvas, so you should attempt to cut an area of the same general shape as the icon. The ratio of window icon width to height is 64:50.

You can use the file obtained with *icut* as a window icon file, and 4Sight will scale and dither it on the fly. However, to increase efficiency and image quality, you may want to scale it yourself. To do so, first run the image tool *istat* on the *icut* file:

```
istat foo
```

istat gives a readout of various image statistics; the important ones for scaling are the first two values, the image width (*xsize*) and image height (*ysize*). The dimension of stowed windows is 50 NeWS (4Sight) points high by 64 NeWS points wide. The ratio of screen pixels to points is 4:3; this yields the following scaling factors:

$xscale = 85.33/(xsize)$
$yscale = 66.67/(ysize)$

To scale the image, use the image tool *izoom*. *izoom* takes in input file, an output file, and width and height scaling factors as arguments. To scale an image file *foo* whose dimensions are 620 pixels wide by 500 pixels high and make it into a console window icon, you would type the following on the command line:

```
izoom foo $HOME/.4sight/icons/console.icon .137 .133
```

Note: Some programs do not use this mechanism to draw their window icons; specifically, those that draw their own icons rather than let 4Sight do it for them. Certain NeWS programs, such as the Calculator draw their own icons in PostScript; Graphics Library programs using `iconsize` to draw their icons use Graphics Library commands to do so. Both of these methods override the window icon mechanism described here.

4.6 Redefining Window Manager Functions

In previous chapters of this section, and in the NeWS section of this manual, you have seen examples of redefining Window Manager functions to change the look and feel of the windowing environment. The most appropriate place to make such changes is in the *user.ps* file in your home directory.

Most of the useful Window Manager functions can be found in files located in the directory */usr/NeWS/lib/NeWS*. The most interesting of these files are described in Appendix A. Although it may be tempting to edit these files directly, **you should not attempt to directly edit any file in /usr/NeWS/lib/NeWS**. Even a minor error may cause the NeWS server to hang. Editing these files directly may also create fatal incompatibilities with future software releases.

If you are experienced with NeWS or PostScript and wish to attempt a large-scale rewrite of the Window Manager, you can create local versions of the Window Manager PostScript files by putting them in a directory called *NeWS* in your home directory. You can copy files from */usr/NeWS/lib/NeWS* into this directory, and 4Sight will load these instead of the standard ones. Similar warnings to the above regarding compatibility apply to these custom files; be sure to remove any local *NeWS* files before loading a software upgrade.

Serious NeWS programmers should consider attaching an ASCII terminal to their IRIS so that they have ready access to the workstation if the NeWS server should hang the console. See Section 2, Chapter 2 of this manual, ''Interacting with NeWS'' for information on how to recover if the NeWS server hangs.

/usr/people/4Dgifts contains some examples of modified Window Manager functions. Log in from the startup window as '4Dgifts' to see how the modified files affect the look and feel of the window manager.

Appendix A: IRIS Window Manager PostScript Files

The following NeWS files comprise the IRIS Window Manager. These files reside in */usr/NeWS/lib/NeWS* You can redefine or replace any of the procedures in those files, either globally when the server starts up, or within a PostScript application; the method is described in Chapter 4.

The following sections outline the organization and contents of the set of Window Manager PostScript files.

A.1 Initialization Files

The following files are loaded and executed when 4Sight is started (when a user logs in at the large console window).

init.ps

init.ps initializes the frame buffer, defines a group of C operations in PostScript, defines and starts the 4Sight (NeWS) server, sets constants and system defaults, and loads the other initialization files described below. Before loading these files from /usr/NeWS/lib/NeWS, the server looks for a directory named ''NeWS'' in your home directory. This allows you to create customized versions of these files without altering the originals. *init.ps* also looks for the file *user.ps* in the current directory. This file can contain customization to any Window Manager routine or definition, and can also be used to set up your desktop in a preferred manner. *init.ps* also loads a default set of toolchests from which you can invoke useful programs and window functions. You can add your own toolchests or redefine existing ones; see Chapter 5 for more information on toolchests.

systoklst.ps

systoklst.ps contains a list of PostScript primitives that are contained in the system dictionary **systemdict**.

cursor.ps

cursor.ps builds a dictionary useful for naming characters in cursorfont, a special font of cursors. Client-defined cursor fonts can also be built; see Section 2, Appendix D, ''Font Tools'', for more information.

icon.ps

icon.ps builds a dictionary useful for naming characters in iconfont, a special font of icons.

util.ps

util.ps contains procedures shared between Window Manager user interface packages; anything that is used in more than one package should be defined in here.

sgimenu.ps

Defines the basic menu package used by the Window Manager. The contents of this file are covered in Chapter 2, ''Window and Menu Packages''.

pupmenu.ps

Defines a subclass of the menus defined in *sgimenu.ps* that supports ''*mex*-style'' pop-up menus.

litewin.ps

litewin.ps contains the basic window package for the IRIS Window Manager. It is expanded upon by the file *sgiwin.ps*.

sgiwin.ps

sgiwin.ps contains a subclass of the windows defined in *litewin.ps*. *sgiwin.ps* also contains the ''*mex*-style'' programming interface.

liteUI.ps, mex_foc.ps, UI.ps

These files set up the Window Manager input focus style and support the IRIS-4D keyboard.

A.2 Other PostScript Files

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

compat.ps

Defines routines that make the server backwards-compatible with older NeWS client programs; in effect the server is programmed to emulate previous versions of the server.

liteitem.ps

A simple item package used by the NeWS Item Demo.

litetext.ps

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

debug.ps

PostScript procedures used when debugging.

colors.ps

Implements X.10V4 color set. Adds the dictionary /**Colordict** to the systemdict.

statusdict.ps

Adds a **statusdict** to **systemdict** for extreme printer compatibility. See Section D.3 of the PostScript Language Reference Manual. Many operators are pseudo-implemented, as they have no meaning in a window system.

journal.ps

A package for recording user actions and replaying them in "player-piano" mode, used by the Journal toolchest (see *journalchest*(1W)).

A.3 PostScript Files for User Settings

init.ps searches for these in the directory you started the NeWS server from, and failing that, looks in */usr/NeWS/lib.*

user.ps

Private definitions and re-definitions of system and package PostScript words. *user.ps* is described more fully in Section 2, Chapter 2 of this manual, ''Interacting With NeWS''.

startup.ps

If *startup.ps* exists, *init.ps* will execute the PostScript in it before it loads any of the other packages. This lets you modify characteristics of the server that are used before any of the other PostScript files (including *user.ps*) are loaded; for example, **verbose?** mode and the **InitPaintRoot** procedure that draws the background on the framebuffer while NeWS is loading.

Index

Section 4:

4Sight Manual Pages

Document Version 3.1

4Sight Manual Pages

This section contains IRIX manual page entries for the Graphics Library and DGL Interfaces, the Font Manager, and NeWS.

Almost all Graphics Library subroutines can be used with the Distributed Graphics Library (see Section 1, Chapter 6, ''Using the Distributed Graphics Library'' for exceptions). The complete set of GL subroutines can be found in the *Graphics Library Reference Manual.* If you are using a language other than C, the language-specific manual pages for the GL and DGL subroutines listed below can also be found in the appropriate language edition of the *Graphics Library Reference Manual.*

The tables below enumerate the manual pages found in this section.

GL Interface Manual Pages		
addtopup(3G)	*maxsize*(3G)	*textcolor*(3G)
defpup(3G)	*minsize*(3G)	*textinit*(3G)
dopup(3G)	*newpup*(3G)	*tpon*(3G)
endfullscrn(3G)	*noborder*(3G)	*winclose*(3G)
foreground(3G)	*noport*(3G)	*winconstraints*(3G)
freepup(3G)	*pagecolor*(3G)	*windepth*(3G)
fudge(3G)	*prefposition*(3G)	*winget*(3G)
fullscrn(3G)	*prefsize*(3G)	*winmove*(3G)
getorigin(3G)	*reshapeviewport*(3G)	*winopen*(3G)
getsize(3G)	*screenspace*(3G)	*winpop*(3G)
getwscrn(3G)	*scrnattach*(3G)	*winposition*(3G)
iconsize(3G)	*scrnselect*(3G)	*winpush*(3G)
icontitle(3G)	*setpup*(3G)	*winset*(3G)
imakebackground(3G)	*stepunit*(3G)	*wintitle*(3G)
keepaspect(3G)	*swinopen*(3G)	

Table 1. GL Interface Manual Pages

DGL Interface Manual Pages	
dglclose(3G)	*finish*(3G)
dglopen(3G)	*gflush*(3G)

Table 2. DGL Interface Manual Pages

Font Manager Manual Pages		
fmcachedisable(3W)	*fmgetchrwidth*(3W)	*fmmatrix*(3W)
fmcacheenable(3W)	*fmgetcomment*(3W)	*fmoutchar*(3W)
fmcachelimit(3W)	*fmgetfontinfo*(3W)	*fmprintermatch*(3W)
fmenumerate(3W)	*fmgetfontname*(3W)	*fmprstr*(3W)
fmfindfont(3W)	*fmgetstrwidth*(3W)	*fmscalefont*(3W)
fmfontpath(3W)	*fmgetwholemetrics*(3W)	*fmsetcachelimit*(3W)
fmfreefont(3W)	*fminit*(3W)	*fmsetfont*(3W)
fmgetchacheused(3W)	*fmmakefont*(3W)	*fmsetpath*(3W)

Table 3. Font Manager Manual Pages

NeWS Manual Pages	
bldfamily(1)	*psterm*(1)
cps(1)	*psview*(1)
dumpfont(1)	*say*(1)
newshost(1)	*sc*(1)
psh(1)	*setnewshost*(1)
psio(3)	*winicons*(5W)

Table 4. NeWS Manual Pages

GL/DGL

Document Version 3.1

NAME

addtopup – adds items to an existing pop-up menu

C SPECIFICATION

void addtopup(pup, str, arg)
long pup;
String str;
long arg;

PARAMETERS

pup expects the menu identifier of the menu to which you want to add. The menu identifier is the returned function value of the menu creation call to either **newpup** or **defpup** functions.

str expects a pointer to the text that you want to add as a menu item. In addition, you have the option of pairing an "item type" flag with each menu item. There are seven menu item type flags:

%t	marks item text as the menu title string.
%F	invokes a routine for every selection from this menu except those marked with a %n. You must specify the invoked routine in the *arg* parameter. The value of the menu item is used as a parameter of the executed routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the function specified by %F.
%f	invokes a routine when this particular menu item is selected. You must specify the invoked routine in the *arg* parameter. The value of the menu item is passed as a parameter of the routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the routine specified by %f. If you have also used the %F flag within this menu, then the result of the %f routine is passed as a parameter of the %F routine.
%l	adds a line under the current entry. You can use this as a visual cue to group like entries together.

%m pops up a menu whenever this menu item is selected. You must provide the menu identifier of the new menu in the *arg* parameter.

%n like %f, this flag invokes a routine when the user selects this menu item. However, %n differs from %f in that it ignores the routine (if any) specified by %F. The value of the menu item is passed as a parameter of the executed routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the function specified by %f.

%x*n* assigns a numeric value to this menu item. This values overrides the default position-based value assigned to this menu item (e.g., the third item is 3). You must enter the numeric value as the *n* part of the text string. Do not use the *arg* parameter to specify the numeric value.

NOTE: If you use the vertical bar delimiter, "|", you can specify multiple menu items in a text string. However, because there is only one *arg* parameter, the text string can contain no more than one item type that references the *arg* parameter.

arg expects the command or submenu that you want to assign to the menu item. You can have only one *arg* parameter for each call to **addtopup**.

DESCRIPTION

addtopup adds items to the bottom of an existing pop-up menu. You can build a menu by using a call to **newpup** to create a menu, followed by a call to **addtopup** for each menu item that you want to add to the menu. To activate and display the menu, submit the menu to **dopup**.

EXAMPLE

This example creates a menu with a submenu:

```
submenu = newpup();
addtopup(submenu, "rotate %f", dorota);
addtopup(submenu, "translate %f", dotran);
addtopup(submenu, "scale %f", doscal);
menu = newpup();
addtopup(menu, "sample %t", 0);
addtopup(menu, "persp", 0);
addtopup(menu, "xform %m", submenu);
addtopup(menu, "greset %f", greset);
```

Because neither the "sample" menu title nor the "persp" menu item refer to the *arg* parameter, you can group "sample", "persp", and "xform" in a single call:

```
addtopup(menu, "sample %t | persp | xform %m", submenu);
```

SEE ALSO

defpup, dopup, freepup, newpup

NOTES

This routine is available only in immediate mode.

When using the Distributed Graphics Library (DGL), you can not call other DGL routines within a function that is called by a popup menu, i.e. a function given as the argument to a %f or %F item type.

NAME

defpup – defines a menu

C SPECIFICATION

long defpup(str [, args ...])
String str;
long args;

PARAMETERS

str expects a pointer to the text that you want to add as a menu item. In addition, you have the option of pairing an ''item type'' flag with each menu item. There are seven menu item type flags:

%t marks item text as the menu title string.

%F invokes a routine for every selection from this menu except those marked with a %n. You must specify the invoked routine in the *arg* parameter. The value of the menu item is used as a parameter of the executed routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the function specified by %F.

%f invokes a routine when this particular menu item is selected. You must specify the invoked routine in the *arg* parameter. The value of the menu item is passed as a parameter of the routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the routine specified by %f. If you have also used the %F flag within this menu, then the result of the %f routine is passed as a parameter of the %F routine.

%l adds a line under the current entry. This is useful in providing visual clues to group like entries together.

%m pops up a menu whenever this menu item is selected. You must provide the menu identifier of the new menu in the *arg* parameter.

%**n** like %f, this flag invokes a routine when the user selects this menu item. However, %n differs from %f in that it ignores the routine (if any) specified by %F. The value of the menu item is passed as a parameter of the executed routine. Thus, if you select the third menu item, the system passes 3 as a parameter to the function specified by %f.

%x*n* assigns a numeric value to this menu item. This values overrides the default position-based value assigned to this menu item (e.g., the third item is 3). You must enter the numeric value as the *n* part of the text string. Do not use the *arg* parameter to specify the numeric value.

args an optional set of arguments. Each argument expects the command or submenu that you want to assign to this menu item. You can use as many *args* parameters as you need.

FUNCTION RETURN VALUE

The returned value for the function is the menu identifier of the menu just defined.

DESCRIPTION

defpup defines a pop-up menu under the window manager and returns a positive menu identifier as the function value.

EXAMPLES

Examples best illustrate the use of the item types.

```
menu = defpup("menu %t|item 1|item 2|item 3|item 4");
```

defines a pop-up menu with title `menu` and four items. You can use a menu of this type as follows:

```
switch (dopup(menu)) {
    case 1: /* item 1 */
        handling code
        break;
    case 2: /* item 2 */
        handling code
        break;
    case 3: /* item 3 */
        handling code
        break;
    case 4: /* item 4 */
        handling code
        break;
}
```

A more complex example is:

```
String str = "menu2 %t %F|1 %n%l|2 %m|3 %f|4 %x234";

menu2 = defpup(str, menufunc, submenu, func);
```

The code defines a menu with title *menu2* and four items with a line under the first one. The code:

```
menuval = dopup(menu2);
```

invokes the menu. Selecting menu item 1 causes **dopup** to return `menufunc(1)`. Rolling off menu item 2 displays *submenu*. **dopup** returns `menufunc(dopup(submenu))` if the user selects from *submenu*. Otherwise, *submenu* disappears and selections are made from *menu*. Buttoning item 3 executes *func* with 3 as its argument, i.e., **dopup** returns `menufunc(func(3))`. Buttoning item 4 causes **dopup** to return `menufunc(234)`. If no item is selected, then **dopup** returns –1.

SEE ALSO

addtopup, dopup, freepup, newpup

NOTES

This routine is available only in immediate mode.

When using the Distributed Graphics Library (DGL), you can not call other DGL routines within a function that is called by a popup menu, i.e. a function given as the argument to a %f or %F item type.

NAME

dglclose – closes the DGL server connection

C SPECIFICATION

void dglclose(sid)
long sid;

PARAMETERS

sid expects the identifier of the server you want to close. If *sid* is negative, then all graphics server connections are closed. Server identifiers are returned by **dglopen**.

DESCRIPTION

dglclose closes the connection to the graphics server associated with the server identifier *sid,* killing the Distributed Graphics Library (DGL) server process and all its windows. If *sid* is negative, then all graphics server connections are closed. Call **dglclose** after **gexit** or when the graphics server is no longer needed. Closing the connection frees up resources on the graphics server.

After a connection is closed, there is no current graphics window and no current graphics server. Calling any routines other than **dglopen, dglclose** or routines that take graphics window identifiers as input parameters will result in an error.

SEE ALSO

dglopen

4Sight User's Guide, ''Using the GL/DGL Interfaces''.

NOTE

This routine is available only in immediate mode.

NAME

dglopen – opens a DGL connection to a graphics server

C SPECIFICATION

long dglopen(svname, type)
String svname;
long type;

PARAMETERS

svname expects a pointer to the name of the graphics server to which you want to open a connection.

For a successful connection, the username on the server must be equivalent (in the sense of *rlogin*(1C)) to the originating account; no provision is made for specifying a password. The remote username used is the same as the local username unless you specify a different remote username. To specify a different remote username, the *svname* string should use the format *username@servername.*

For DECnet connections, if the server account has a password, this password must be specified using the format *username password@servername.* This password is used only for opening the DECnet connection; the two accounts must still be equivalent in the *rlogin* sense.

type expects a symbolic constant that specifies the kind of connection. There are three defined constants for this parameter:

DGLLOCAL indicates a direct connection to the local graphics hardware.

DGLTSOCKET indicates a remote connection via TCP/IP.

DGL4DDN indicates a remote connection via DECnet.

FUNCTION RETURN VALUE

If the connection succeeds, the returned value of the function is a non-negative integer, *serverid*, that identifies the graphics server. If the connection failed, the returned value for the function is a negative integer.

The absolute value of a negative returned value is either a standard error value (defined in <*errno.h*>) or one of several error returns associated specifically with **dglopen**:

ENODEV	*type* is not a valid connection type.
EACCESS	login incorrect or permission denied.
EMFILE	too many graphics connections are currently open.
EBUSY	only one DGLLOCAL connection allowed.
ENOPROTOOPT	DGL service not found in **/etc/services**.
ERANGE	invalid or unrecognizable number representation.
EPROTONOSUPPORT	DGL version mismatch.
ESRCH	the window manager is not running on the server.

DESCRIPTION

dglopen opens a Distributed Graphics Library (DGL) connection to a graphics server (*svname*). After a connection is open, all graphics input and output are directed to that connection. Graphics input and output continue to be directed to the connection until either the connection is closed, another connection is opened or a different connection is selected. A different connection can be selected by calling a subroutine that takes a graphics window identifier as an input parameter, eg. **winset**. The server connection associated with that graphics window identifier becomes the current connection. To close a DGL connection, call **dglclose** with the server identifier returned by **dglopen**.

SEE ALSO

dglclose, finish, gflush, winopen, winset

rlogin(1C) in the *IRIS-4D User's Reference Manual*

4Sight User's Guide, ''Using the GL/DGL Interfaces''.

NOTES

This routine is available only in immediate mode.

This routine is available in both the DGL and GL library. However, only a **DGLLOCAL** connection type is supported by the GL library.

NAME

dopup – displays the specified pop-up menu

C SPECIFICATION

long dopup(pup)
long pup;

PARAMETERS

pup expects the identifier of the pop-up menu you want to display.

FUNCTION RETURN VALUE

The returned value of the function is the value of the item selected from the pop-up menu. If the user makes no menu selection, the returned value of the function is –1.

DESCRIPTION

dopup displays the specified pop-up menu until the user makes a selection. If the calling program has the input focus, the menu is displayed and **dopup** returns the value resulting from the item selection. The value can be returned by a submenu, a function, or a number bound directly to an item. If no selection is made, **dopup** returns –1.

When you first define the menu (using **defpup** or **addtopup**) you specify the list of menu entries and their corresponding actions. See **addtopup** for details.

SEE ALSO

addtopup, defpup, freepup, newpup

NOTE

This routine is available only in immediate mode.

NAME

endfullscrn – ends full-screen mode

C SPECIFICATION

void endfullscrn()

PARAMETERS

none

DESCRIPTION

endfullscrn ends full-screen mode and returns the screenmask and viewport to the boundaries of the current graphics window. **endfullscrn** leaves the current transformation unchanged.

SEE ALSO

fullscrn

NOTE

This routine is available only in immediate mode.

NAME

finish – blocks until the Geometry Pipeline is empty

C SPECIFICATION

void finish()

PARAMETERS

none

DESCRIPTION

finish forces all unsent commands down the Geometry Pipeline to the rendering subsystem followed by a final token. It blocks the calling process until an acknowledgement is returned from the rendering subsystem that the final token has been received.

SEE ALSO

gflush

NOTE

This routine is available only in immediate mode.

NAME

foreground – prevents a graphical process from being put into the background

C SPECIFICATION

void foreground()

PARAMETERS

none

DESCRIPTION

winopen normally runs a process in the background. Call **foreground** before calling **winopen**. It keeps the process in the foreground, so that you can interact with it from the keyboard. When the process is in the foreground, it interacts in the usual way with the IRIX input/output routines.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

NAME

freepup – deallocates a menu

C SPECIFICATION

void freepup(pup)
long pup;

PARAMETERS

pup expects the menu identifier of the pop-up menu that you want to deallocate.

DESCRIPTION

freepup deallocates a pop-up menu, freeing the memory reserved for its data structures.

SEE ALSO

defpup, addtopup, dopup, newpup

NOTE

This routine is available only in immediate mode.

NAME

fudge – specifies fudge values that are added to a graphics window

C SPECIFICATION

void fudge(xfudge, yfudge)
long xfudge, yfudge;

PARAMETERS

xfudge	expects the number of pixels added in the *x* direction.
yfudge	expects the number of pixels added in the *y* direction.

DESCRIPTION

fudge specifies fudge values that are added to the dimensions of a graphics window when it is sized. Typically, you use it to create interior window borders. Call **fudge** prior to calling **winopen**.

fudge is useful in conjunction with **stepunit** and **keepaspect**. With **stepunit** the window size for integers *m* and *n* is:

$$width = xunit \times m + xfudge$$
$$height = yunit \times n + yfudge$$

With **keepaspect** the window size is (*width*, *height*), where:

$$(width - xfudge) \times yaspect = (height - yfudge) \times xaspect$$

SEE ALSO

keepaspect, stepunit, winopen

NOTE

This routine is available only in immediate mode.

NAME

fullscrn – gives a program the entire screen as a window

C SPECIFICATION

void fullscrn()

PARAMETERS

none

DESCRIPTION

fullscrn gives a program the entire screen as a window. It calls:

```
viewport(0, getgdesc(GD_XPMAX), 0, getgdesc(GD_YPMAX))
```

and **ortho2** to set up an orthographic projection that maps world coordinates to screen coordinates. **fullscrn** eliminates all protections that prevent graphics processes from drawing on each other. Use it with caution or a sense of humor.

SEE ALSO

endfullscrn, winopen

NOTE

This routine is available only in immediate mode.

NAME

getorigin – returns the position of a graphics window

C SPECIFICATION

void getorigin(x, y)
long *x, *y;

PARAMETERS

x expects a pointer to the location into which the system should copy the *x* position (in pixels) of the lower left corner of the graphics window.

y expects a pointer to the location into which the system should copy the *y* position (in pixels) of the lower left corner of the graphics window.

DESCRIPTION

getorigin returns the position (in pixels) of the lower-left corner of a graphics window. Call **getorigin** after graphics initialization.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

NAME

getsize – returns the size of a graphics window

C SPECIFICATION

void getsize(x, y)
long *x, *y;

PARAMETERS

x expects a pointer to the location into which the system should copy the width (in pixels) of a graphics window.

y expects a pointer to the location into which the system should copy the height (in pixels) of a graphics window.

DESCRIPTION

getsize gets the dimensions (in pixels) of the graphics window used by a graphics program. Call **getsize** after **winopen**.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

NAME

getwscrn – returns the screen upon which the current window appears

C SPECIFICATION

long getwscrn()

PARAMETERS

none

FUNCTION RETURN VALUE

The returned function value is the screen number upon which the current window appears.

DESCRIPTION

getwscrn gets the screen number of the current window.

NOTE

This routine is available only in immediate mode.

NAME

gflush – flushs the DGL client buffer

C SPECIFICATION

void gflush()

PARAMETERS

none

DESCRIPTION

gflush has no function in the Graphics Library, but is included to provide compatibility with Distributed Graphics Library (DGL).

SEE ALSO

finish

4Sight User's Guide, ''Using the GL/DGL Interfaces''.

NOTE

The DGL on the client buffers the output from most graphics routines for efficient block transfer to the server. The DGL version of **gflush** sends all buffered but untransmitted graphics data to the server. Certain graphics routines, notably those that return values, also flush the client buffer when they execute.

NAME

iconsize – specifies the icon size of a window

C SPECIFICATION

void iconsize(x,y)
long x, y;

PARAMETERS

x expects the width (in pixels) for the icon.

y expects the height (in pixels) for the icon.

DESCRIPTION

iconsize specifies the size (in pixels) of the window used to replace a stowed window. If a window has an icon size, the window manager will re-shape the window to be that size and send a **REDRAWICONIC** token to the graphics queue when the user stows that window. Your code can use an event loop to test for this token and can call graphics library subroutines to draw the icon for the stowed window. Windows without an icon size are handled by the window manager with the locally appropriate default behavior.

To assign a new window an icon size, call **iconsize** before you open the window. To give an existing window an icon size, use **iconsize** with **winconstraints**.

SEE ALSO

qdevice, winconstraints, winopen

NOTES

This routine is available only in immediate mode.

Any application using **iconsize** should also call **qdevice** to queue the tokens **WINFREEZE** and **WINTHAW** after opening the window.

NAME

icontitle – assigns the icon title for the current graphics window.

C SPECIFICATION

void icontitle(name)
String name;

PARAMETERS

name expects a pointer to the string containing the icon title.

DESCRIPTION

icontitle specifies the string displayed on an icon if the window manager draws that window's icon.

SEE ALSO

iconsize

NAME

imakebackground – registers the screen background process

C SPECIFICATION

void imakebackground()

PARAMETERS

none

DESCRIPTION

imakebackground registers a process that maintains the screen background. Call it before **winopen**. The process should redraw the screen background each time it receives a REDRAW event.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

NAME

keepaspect – specifies the aspect ratio of a graphics window

C SPECIFICATION

void keepaspect(x, y)
long x, y;

PARAMETERS

x expects the horizontal proportion of the aspect ratio.

y expects the vertical proportion of the aspect ratio.

DESCRIPTION

keepaspect specifies the aspect ratio of a graphics window. Call it at the beginning of a graphics program. It takes effect when you call **winopen**. The resulting graphics window maintains the aspect ratio specified in **keepaspect**, even if it changes size.

For example, `keepaspect(1, 1)` always results in a square graphics window. You can also call **keepaspect** in conjunction with **winconstraints** to modify the enforced aspect ratio after the window has been created.

SEE ALSO

fudge, winconstraints, winopen

NOTE

This routine is available only in immediate mode.

NAME

maxsize – specifies the maximum size of a graphics window

C SPECIFICATION

void maxsize(x, y)
long x, y;

PARAMETERS

x expects the maximum width of a graphics window. The width is measured in pixels.

y expects the maximum height of a graphics window. The height is measured in pixels.

DESCRIPTION

maxsize specifies the maximum size (in pixels) of a graphics window. Call it at the beginning of a graphics program before **winopen**. **maxsize** takes effect when **winopen** is called.

You can also call **maxsize** in conjunction with **winconstraints** to modify the enforced maximum size after the window has been created. The default maximum size is `getgdesc(GD_XPMAX)` pixels wide and `getgdesc(GD_YPMAX)` pixels high. The user can reshape the graphics window, but the window manager does not allow it to become larger than the specified maximum size.

SEE ALSO

getgdesc, minsize, winopen

NOTE

This routine is available only in immediate mode.

NAME

minsize – specifies the minimum size of a graphics window

C SPECIFICATION

void minsize(x, y)
long x, y;

PARAMETERS

x expects the minimum width of a graphics window. The width is measured in pixels. The lowest legal value for this parameter is 1.

y expects the minimum height of a graphics window. The height is measured in pixels. The lowest legal value for this parameter is 1.

DESCRIPTION

minsize specifies the minimum size (in pixels) of a graphics window. Call it at the beginning of a graphics program. It takes effect when **winopen** is called. You can also call **minsize** with **winconstraints** to modify the enforced minimum size after the window has been created. The default minimum size is 40 pixels wide and 30 pixels high. You can reshape the window, but the window manager does not allow it to become smaller than the specified minimum size.

SEE ALSO

maxsize, winopen

NOTE

This routine is available only in immediate mode.

NAME

newpup – allocates and initializes a structure for a new menu

C SPECIFICATION

long newpup()

PARAMETERS

none

FUNCTION RETURN VALUE

The returned value of this function is a menu identifier.

DESCRIPTION

newpup allocates and initializes a structure for a new menu; it returns a positive menu identifier. Use **newpup** with **addtopup** to create pop-up menus.

SEE ALSO

addtopup, defpup, dopup, freepup

NOTE

This routine is available only in immediate mode.

NAME

noborder – specifies a window without any borders

C SPECIFICATION

void noborder()

PARAMETERS

none

DESCRIPTION

noborder specifies a window that has no borders around its drawable area. Call **noborder** before you open the window.

SEE ALSO

winconstraints

NAME

noport – specifies that a program does not need screen space

C SPECIFICATION

noport()

PARAMETERS

none

DESCRIPTION

noport specifies that a graphics program does not need screen space, and therefore does not need a graphics window. This is useful for programs that only read or write the color map. Call **noport** at the beginning of a graphics program; then call **winopen** to do a graphics initialization.

The system ignores **noport** if **winopen** is not called.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

NAME

pagecolor – sets the color of the textport background

C SPECIFICATION

void pagecolor(pcolor)
Colorindex pcolor;

PARAMETERS

pcolor expects an index into the current color map.

DESCRIPTION

pagecolor sets the background color of the textport of the calling process. If the calling process was invoked from a *wsh* window, this window is used for its textport; otherwise, the process does not have a textport and this routine does nothing.

SEE ALSO

textcolor

wsh(1) in the *User's Reference Manual.*

NOTES

This routine is available only in immediate mode.

A process launched from *4Sight* or *The IRIS WorkSpace*™ will not have a textport. Therefore, we do not recommend the use of this routine in new development.

NAME

prefposition – specifies the preferred location and size of a graphics window

C SPECIFICATION

void prefposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

PARAMETERS

x1 expects the x coordinate position (in pixels) of the point at which one corner of the window is to be.

x2 expects the x coordinate position (in pixels) of the point at which the opposite corner of the window is to be.

y1 expects the y coordinate position (in pixels) of the point at which one corner of the window is to be.

y2 expects the y coordinate position (in pixels) of the point at which the opposite corner of the window is to be.

DESCRIPTION

prefposition specifies the preferred location and size of a graphics window. You specify the location in pixels (*x1*, *x2*, *y1*, y2). Call **prefposition** at the beginning of a graphics program. Use **prefposition** with **winconstraints** to modify the enforced size and location after the window has been created. Calling **winopen** activates the constraints specified by **prefposition**. If **winopen** is not called, **prefposition** is ignored.

SEE ALSO

winconstraints, winopen

NOTE

This routine is available only in immediate mode.

NAME

prefsize – specifies the preferred size of a graphics window

C SPECIFICATION

void prefsize(x, y)
long x, y;

PARAMETERS

x expects the width of the graphics window. The width is measured in pixels.

y expects the height of the graphics window. The height is measured in pixels.

DESCRIPTION

prefsize specifies the preferred size of a graphics window as *x* pixels by *y* pixels. Call **prefsize** at the beginning of a graphics program.

Once a window is created, you must use **prefsize** with **winconstraints** in order to modify the enforced window size. Calling **winopen** activates the constraints specified by **prefsize.** If **winopen** is not called, **prefsize** is ignored.

SEE ALSO

winconstraints, winopen

NOTE

This routine is available only in immediate mode.

NAME

reshapeviewport – sets the viewport to the dimensions of the current graphics window

C SPECIFICATION

void reshapeviewport()

PARAMETERS

none

DESCRIPTION

reshapeviewport sets the viewport to the dimensions of the current graphics window.

reshapeviewport is equivalent to:

```
long xsize, ysize;

getsize(&xsize, &ysize);
viewport(0, xsize-1, 0, ysize-1);
```

Use **reshapeviewport** when REDRAW events are received. It is most useful in programs that are independent of the size and shape of the viewport.

SEE ALSO

getorigin, getsize, viewport

NOTE

This routine is available only in immediate mode.

NAME

screenspace – interpret graphics positions as absolute screen coordinates

C SPECIFICATION

void screenspace()

PARAMETERS

none

DESCRIPTION

The origin, in screen coordinates, is at the lower left corner of the screen. In window coordinates the origin is at the lower left corner of the user-defined graphics window. **screenspace** instructs a program to interpret graphics positions as absolute screen coordinates. This allows pixels and locations outside a program's graphics window to be read.

screenspace is equivalent to:

```
long xmin, ymin;


getorigin(&xmin, &ymin);
viewport(-xmin, getgdesc(GD_XPMAX)-xmin, -ymin, getgdesc(GD_YPMAX)-ymin);
ortho2(-0.5, getgdesc(GD_XPMAX)+0.5, -0.5, getgdesc(GD_YPMAX)+0.5);
```

SEE ALSO

getorigin, viewport, ortho2

NOTE

This routine is available only in immediate mode.

NAME

scrnattach – attaches the input focus to a screen

C SPECIFICATION

long scrnattach(gsnr)
long gsnr;

PARAMETERS

gsnr expects a screen number.

FUNCTION RETURN VALUE

The function returns the screen that previously had input focus, or –1 if there was an error (e.g., *gsnr* is not a valid screen number).

DESCRIPTION

scrnattach attaches the input focus to the specified screen. It waits for any window manager or menu interaction to be completed before doing the attach.

Use `getgdesc(GD_NSCRNS)` to determine the number of screens available to your program. Screens are numbered starting from zero. When the user submits an erroneous value, INFOCUSSCRN is unchanged. Use `scrnselect(INFOCUSSCRN)` to obtain the screen that currently has the input focus.

SEE ALSO

getgdesc, getwscrn, scrnselect

NOTE

This routine is available only in immediate mode.

NAME

scrnselect – selects the screen upon which new windows are placed

C SPECIFICATION

long scrnselect(gsnr)
long gsnr;

PARAMETERS

gsnr expects a screen number or the symbolic constant, INFOCUSSCRN, to select the screen with the input focus at the time the command is issued.

FUNCTION RETURN VALUE

The function returns the previously selected screen, or –1 if there was an error (e.g., *gsnr* is not a valid screen number).

DESCRIPTION

scrnselect selects the screen upon which a subsequent **winopen ginit gbegin** creates a window. It also selects the screen to which **getgdesc** inquiries refer. You can call **scrnselect** prior to graphics initialization.

Use `getgdesc(GD_NSCRNS)` to determine the number of screens available to your program. Screens are numbered starting from zero. On error **scrnselect** leaves the selected screen unchanged.

SEE ALSO

getgdesc, ginit, scrnattach, winopen

NOTE

This routine is available only in immediate mode.

NAME

setpup – sets the display characteristics of a given pop up menu entry

C SPECIFICATION

void setpup(pup, entry, mode)
long pup, entry;
unsigned long mode;

PARAMETERS

pup	expects the menu identifier of the menu whose entries you want to change. The menu identifier is the returned function value of the menu creation call to either **newpup** or **defpup**.
entry	expects the position of the entry in the menu, indexed from 1.
mode	expects a symbolic constant that indicates the display characteristics you want to apply to the chosen entry. For this parameter there are two defined symbolic constants:

PUP_NONE, no special display characteristics, fully functional if selected. This is the default mode for newly created menu entries.

PUP_GREY, entry is greyed-out and disabled. Selecting a greyed-out entry has the same behavior as selecting the title bar. If the greyed-out entry has a submenu associated with it, that submenu does not display.

DESCRIPTION

Use **setpup** to alter the display characteristics of a pop up menu entry. Currently, you use this routine to disable and grey-out a menu entry.

EXAMPLE

Here is an example that disables a single entry:

```
menu = newpup();
addtopup(menu,"menu %t |item 1 |item 2 |item 3 |item 4",0);
setpup(menu, 1, PUP_GREY);
```

Subsequent calls of `dopup(menu)` would display the menu with the menu entry labeled "item 1" is greyed out, and never gets a return value of 1.

SEE ALSO

defpup, dopup, freepup, newpup

NOTE

This routine is available only in immediate mode.

NAME

stepunit – specifies that a graphics window change size in discrete steps

C SPECIFICATION

void stepunit(xunit, yunit)
long xunit, yunit;

PARAMETERS

xunit expects the amount of change per unit in the x direction. The amount is measured in pixels.

yunit expects the amount of change per unit in the y direction. The amount is measured in pixels.

DESCRIPTION

stepunit specifies the size of the change in a graphics window in discrete steps of *xunit* and *yunit.* Call **stepunit** at the beginning of a graphics program; it takes effect when you call **winopen**. **stepunit** resizes graphics windows in units of a standard size (in pixels). If **winopen** is not called, **stepunit** is ignored.

SEE ALSO

winopen, fudge

NOTE

This routine is available only in immediate mode.

NAME

swinopen – creates a graphics subwindow

C SPECIFICATION

long swinopen(parent)
long parent;

PARAMETERS

parent expects the GID (graphics window identifier) of the window (or subwindow) in which you want to open a subwindow. The GID is the returned function value of a previous call to either **swinopen** or **winopen.**

FUNCTION RETURN VALUE

The returned value of this function is either a **–1** or the graphics window identifier for the subwindow just created. Use this value to identify this subwindow to other graphics routines.

A returned function value of **–1** indicates that the system cannot create any more graphics windows.

DESCRIPTION

swinopen creates a graphics subwindow. The graphics state of the new subwindow is initialized to its defaults (see **greset)** and it becomes the current window.

Subwindows have no window borders or window manager function buttons. Window constraints do not apply to subwindows. A subwindow is repositioned automatically when its parent is moved, so that its origin with respect to the parent's origin remains constant. Resizing the parent does not automatically resize a subwindow, but keeps the distance between the upper left hand corners constant. Imaging in a subwindow is limited (clipped) to the area of the parent window.

After calling **swinopen,** the application must call **winposition** to specify the location of the subwindow's boundaries with respect to the origin of its parent window.

If *parent* is the GID of a subwindow, the parent of that subwindow becomes the parent of the new subwindow for the purpose of positioning the subwindow.

When using the DGL (Distributed Graphics Library), the graphics window identifier also identifies the graphics server associated with the window.

swinopen queues the pseudo devices INPUTCHANGE and REDRAW.

SEE ALSO

greset, winclose, winget, winopen, winposition, winset

NAME

textcolor – sets the color of text in the textport

C SPECIFICATION

void textcolor(tcolor)
Colorindex tcolor;

PARAMETERS

tcolor expects an index into the current color map.

DESCRIPTION

textcolor sets the color of all the text in the textport of the calling process. If the calling process was invoked from a *wsh* window, this window is used for its textport; otherwise, the process does not have a textport and this routine does nothing.

SEE ALSO

pagecolor

wsh(1) in the *User's Reference Manual.*

NOTES

This routine is available only in immediate mode.

A process launched from *4Sight* or *The IRIS WorkSpace*™ will not have a textport. Therefore, we do not recommend the use of this routine in new development.

NAME

textinit – initializes the textport

C SPECIFICATION

void textinit()

PARAMETERS

none

DESCRIPTION

textinit initializes the textport of the calling process to its default size, location, textcolor, and pagecolor. If the calling process was invoked from a *wsh* window, this window is used for its textport; otherwise, the process does not have a textport and this routine does nothing.

SEE ALSO

pagecolor, textcolor, textport, tpon

wsh(1) in the *User's Reference Manual.*

NOTES

This routine is available only in immediate mode.

A process launched from *4Sight* or *The IRIS WorkSpace*™ will not have a textport. Therefore, we do not recommend the use of this routine in new development.

NAME

tpon, tpoff – control the visibility of the textport

C SPECIFICATION

void tpon()

void tpoff()

PARAMETERS

none

DESCRIPTION

tpon pops the textport of the calling process, bringing it to the front of any windows that conceal it. **tpoff** pushs the textport down behind all other windows, effectively hiding it. If the calling process was invoked from a *wsh* window, this window is used for its textport; otherwise, the process does not have a textport and this routine does nothing.

SEE ALSO

textinit, textport

wsh(1) in the *User's Reference Manual.*

NOTES

This routine is available only in immediate mode.

A process launched from *4Sight* or *The IRIS WorkSpace*^{TM} will not have a textport. Therefore, we do not recommend the use of these routines in new development.

NAME

winclose – closes the identified graphics window

C SPECIFICATION

void winclose(gwid)
long gwid;

PARAMETERS

gwid expects the identifier for the graphics window that you want closed.

DESCRIPTION

winclose closes the graphics window associated with identifier *gwid.* The identifier for a window is the function return value from the call to *winopen* that created the window.

When using the Distributed Graphics Library (DGL), the graphics window identifier also identifies the graphics server associated with the window. The DGL directs all subsequent Graphics Library input and output to the server associated with *gwid.*

If the window being closed is on a screen for which the **getgdesc** inquiry GD_SCRNTYPE returns GD_SCRNTYPE_NOWM, **winclose** leaves the image undisturbed.

SEE ALSO

getgdesc, winopen

NOTE

This routine is available only in immediate mode.

NAME

winconstraints – binds window constraints to the current window

C SPECIFICATION

void winconstraints()

PARAMETERS

none

DESCRIPTION

winconstraints binds the currently specified constraints to the current graphics window. (Logically, because this assumes the existence of a current graphics window, you must have previously called **winopen**.) Prior to calling **winconstraints**, you can set the the values of the window constraints by using the following commands: **minsize**, **maxsize**, **keepaspect**, **prefsize**, **iconsize**, **noborder**, **noport**, **stepunit**, **fudge**, **imakebackground**, and **prefposition**.

After binding these constraints to a window, **winconstraints** resets the window constraints to their default values, if any.

SEE ALSO

fudge, keepaspect, iconsize, imakebackground, maxsize, minsize, noborder, noport, prefposition, prefsize, stepunit, winopen

NOTE

This routine is available only in immediate mode.

NAME

windepth – measures how deep a window is in the window stack

C SPECIFICATION

long windepth(gwid)
long gwid;

PARAMETERS

gwid expects the window identifier for the window you want to test.

FUNCTION RETURN VALUE

The returned value of this function is a number that you can use to determine that stacking order of windows on the screen.

DESCRIPTION

windepth returns a number which can be compared against the **windepth** return value for other windows to determine the stacking order of a programs windows on the screen.

When using the Distributed Graphics Library DGL), the graphics window identifier also identifies the graphics server associated with the window. The DGL directs all subsequent Graphics Library input and output to the server associated with *gwid.*

SEE ALSO

winpush, winpop

NAME

winget – returns the identifier of the current graphics window

C SPECIFICATION

long winget()

PARAMETERS

none

FUNCTION RETURN VALUE

The returned value for this function is the identifier of the current graphics window.

DESCRIPTION

winget returns the identifier of the current graphics window. The current graphics window is the window to which the system directs the output from graphics routines.

SEE ALSO

winset

NOTE

This routine is available only in immediate mode.

NAME

winmove – moves the current graphics window by its lower-left corner

C SPECIFICATION

void winmove(orgx, orgy)
long orgx, orgy;

PARAMETERS

orgx expects the *x* coordinate of the location to which you want to move the current graphics window.

orgy expects the *y* coordinate of the location to which you want to mcve the current graphics window.

DESCRIPTION

winmove moves the current graphics window so that its origin is at the screen coordinates (in pixels) specified by *orgx, orgy*. The origin of the current graphics window is its lower-left corner. **winmove** does not change the size and shape of the window.

SEE ALSO

winposition

NOTE

This routine is available only in immediate mode.

NAME

winopen – creates a graphics window

C SPECIFICATION

long winopen(name)
String name;

PARAMETERS

name expects the window title that is displayed on the left hand side of the title bar for the window. If you do not want a title, pass a zero-length string.

FUNCTION RETURN VALUE

The returned value for this function is the graphics window identifier for the window just created. Use this value to identify the graphics window to other windowing functions. Only the lower 16 bits are significant, since a graphics window identifier is the value portion of a REDRAW event queue entry. If no additional windows are available, this function returns –1.

DESCRIPTION

winopen creates a graphics window as defined by the current values of the window constraints. This new window becomes the current window. If this is the first time that your program has called **winopen**, the system also initializes the Graphics Library.

Except for size and location, the system maintains default values for the constraints on a window. You can change these default window constraints if you call the routines **minsize, maxsize, keepaspect, prefsize, prefposition, stepunit, fudge, iconsize, noborder, noport, imakebackground,** and **foreground** before you call **winopen.** If the a window's size and location (or both) are left unconstrained, the system allows the user to place and size the window.

After a call to **winopen,** the system resets the graphics state of the window (this includes window constraints) to its default value.

When using the Distributed Graphics Library (DGL), the window identifier also identifies the window's graphics server. The DGL directs all graphics input and output to the current window's server; subsequent Graphics Library subroutines are executed by the window's server.

winopen queues the pseudo devices INPUTCHANGE and REDRAW.

SEE ALSO

foreground, fudge, iconsize, imakebackground, keepaspect, minsize, maxsize, noborder, noport, prefsize, prefposition, stepunit, winclose

4Sight User's Guide, ''Using the GL/DGL Interfaces''.

NOTE

This routine is available only in immediate mode.

NAME

winpop – moves the current graphics window in front of all other windows

C SPECIFICATION

void winpop()

PARAMETERS

none

DESCRIPTION

When more than one window tries to occupy the same space on the screen, the system stacks them on top of each other—thus obscuring (either partially or completely) the underlying graphics window or windows.

Use **winpop** to take the current graphics window from anywhere in the stack of windows and place it on top.

SEE ALSO

winpush

NOTE

This routine is available only in immediate mode.

NAME

winposition – changes the size and position of the current graphics window

C SPECIFICATION

void winposition(x1, x2, y1, y2)
long x1, x2, y1, y2;

PARAMETERS

x1 expects the *x* screen coordinate (in pixels) of the first corner of the new location for the current graphics window. The first corner of the new window is the corner diagonally opposite the second corner.

x2 expects the *x* screen coordinate (in pixels) of the second corner of the new location for the current graphics window.

y1 expects the *y* screen coordinate (in pixels) of the first corner of the new location for the current graphics window.

y2 expects the *y* screen coordinate (in pixels) of the second corner of the new location for the current graphics window.

DESCRIPTION

winposition moves and reshapes the current graphics window to match the screen coordinates *x1, x2, y1, y2* (calculated in pixels). This differs from **prefposition** because the reshaped window is not fixed in size and shape, and can be reshaped interactively.

SEE ALSO

prefposition, prefsize, winmove

NOTE

This routine is available only in immediate mode.

NAME

winpush – places the current graphics window behind all other windows

C SPECIFICATION

void winpush()

PARAMETERS

none

DESCRIPTION

When more than one window tries to occupy the same space on the screen, the system stacks them on top of each other—thus obscuring (either partially or completely) the underlying graphics window or windows.

Use **winpush** to take the current graphics window from anywhere in the stack of windows and push it to the bottom.

SEE ALSO

winpop

NOTE

This routine is available only in immediate mode.

NAME

winset – sets the current graphics window

C SPECIFICATION

void winset(gwid)
long gwid;

PARAMETERS

gwid expects a graphics window identifier.

DESCRIPTION

winset takes the graphics window associated with identifier *gwid* and makes it the current window. The system directs all graphics output to the current graphics window.

When using the Distributed Graphics Library (DGL), the graphics window identifier also identifies the graphics server associated with the window. The DGL directs all subsequent Graphics Library input and output to the server associated with *gwid*.

SEE ALSO

winget

NOTE

This routine is available only in immediate mode.

NAME

wintitle – adds a title bar to the current graphics window

C SPECIFICATION

void wintitle(name)
String name;

PARAMETERS

name expects the title you want displayed in the title bar of the current graphics window.

DESCRIPTION

wintitle adds a title to the current graphics window.

Use `wintitle("")` to clear the title.

SEE ALSO

winopen

NOTE

This routine is available only in immediate mode.

Font Manager

Document Version 3.1

NAME

fmcachedisable – disable font cache flushing

SYNOPSIS

#include <fmclient.h>

void fmcachedisable()

DESCRIPTION

The Font Manager provides font cache flushing by default. To disable automatic flushing of the font cache, call *fmcachedisable* . The Font Manager continues to keep track of the space fonts occupy, but does nothing to flush the space when it exceeds the font cache limit. You can turn on font cache flushing again by calling *fmcacheenable* .

SEE ALSO

fmcacheenable(3W), fmcachelimit(3W), fmgetcacheused(3W), fmsetcachelimit(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmcacheenable – enable font cache flushing

SYNOPSIS

#include <fmclient.h>

void fmcacheenable()

DESCRIPTION

The Font Manager provides font cache flushing by default. You can disable font cache flushing with a call to *fmcachedisable.* The Font Manager continues to keep track of the space fonts occupy, but does nothing to flush the space when it exceeds the font cache limit. To restart automatic font cache flushing, call *fmcacheenable* .

SEE ALSO

fmcachedisable(3W), fmcachelimit(3W), fmgetcacheused(3W), fmsetcachelimit(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmcachelimit – return current font cache limit in quanta

SYNOPSIS

#include <fmclient.h>

long fmcachelimit()

DESCRIPTION

The font cache maintains the space reserved for font data. Use *fmcachelimit* to get the number of Font Manager cache quanta currently set as the upper size limit of the font cache. To calculate the number of bytes reserved for the font cache, multiply the returned value of *fmcachelimit* by FMCACHE_QUANTUM. So, if *fmcachelimit* returns a ''4'', and FMCACHE_QUANTUM is 100,000, the cache upper limit size is 400,000. To set the font cache limit, use *fmsetcachelimit.*

SEE ALSO

fmcachedisable(3W), fmcacheenable(3W), fmgetcacheused(3W), fmsetcachelimit(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmenumerate – enumerate the available font faces

SYNOPSIS

#include <fmclient.h>

fmenumerate (callback)
void (*callback)();

DESCRIPTION

fmenumerate accepts a callback procedure as an argument. It calls the procedure once for each font face name in the font directories in the font path, providing a string pointer to the callback routine. For example, the following code prints the name of each "family" to the terminal:

```
void printname(str)
    char *str;
{
        printf("%s\n", str);
}

main()
{
        fminit();
        fmenumerate(printname);
}
```

SEE ALSO

fminit(3W), fmfontpath(3W), fmsetpath(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmfindfont – select a font face.

SYNOPSIS

#include <fmclient.h>

fmfonthandle fmfindfont(face)
char *face;

DESCRIPTION

After initializing the font manager with *fminit()*, this is probably the first routine to be called in order to select a specific font to either image or query. Its functionality parallels the findfont operator of **PostScript.**

fmfindfont's argument is a string specifying the face of the font that later can be (optionally) sized and read from the disk. For example, to prepare to use Times-Roman, call *fmfindfont* with Times-Roman as the argument:

fmfindfont("Times-Roman");

SEE ALSO

fminit(3w), fmfontpath(3w), fmsetpath(3w), fmscalefont(3w), fmsetfont(3w).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmfontpath – get the current font path.

SYNOPSIS

#include <fmclient.h>

char *fmfontpath()

DESCRIPTION

fmfontpath returns a pointer to a string that describes the current search path for finding font files. It is a colon-separated list of directories that originate at the root. The default path is "/usr/lib/fmfonts".

SEE ALSO

fminit(3W), fmsetpath(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmfreefont – free the storage for a font

SYNOPSIS

#include <fmclient.h>

void fmfreefont(fh)
fmfonthandle fh;

DESCRIPTION

fmfreefont frees the storage associated with a font in a given rotation and size. To be sure the correct font/rotation/size instance is freed, the same page matrix must be in force as when the font was originally queried or imaged. Since normal usage of the font manager is not to change the page matrix, this should not pose a great risk, but it should be kept in mind.

SEE ALSO

fminit(3W), fmscalefont(3W), fmgetpagematrix(3W), fmsetpagematrix(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetcacheused – return current font cache limit in quanta

SYNOPSIS

#include <fmclient.h>

long fmgetcacheused()

DESCRIPTION

This routine returns the exact number of bytes used by the font cache. There are no implied multipliers.

SEE ALSO

fmcachedisable(3W), fmcacheenable(3W), fmcachelimit(3W), fmsetcachelimit(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetchrwidth – return the width of a character

SYNOPSIS

#include <fmclient.h>

long fmgetchrwidth(fh, ch)
fmfonthandle fh;
unsigned char ch;

DESCRIPTION

fmgetchrwidth returns the number of pixels the given character moves in the x dimension when imaged. This value is rounded to an integer. If that character glyph does not exist, the width of a space is returned. If a space does not exist, the width of the font is returned.

SEE ALSO

fminit(3w).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetcomment – return a comment associated with a font

SYNOPSIS

#include <fmclient.h>

long fmgetcomment(fh, slen, str);
fmfonthandle fh;
int slen;
char *str;

DESCRIPTION

fmgetcomment returns a comment associated with a font in the space provided by the client in *str.* The length of the passed-in character array is passed in *slen; fmgetcomment* will not write more characters than specified by *slen.* If there is an error in locating the font specified by *fh,* a –1 is returned. If no comment exists for the font, a -1 is returned, otherwise, the string's length is returned.

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetfontinfo – return information about the overall font

SYNOPSIS

#include <fmclient.h>

long fmgetfontinfo(fh, info)
fmfonthandle fh;
fmfontinfo *info;

DESCRIPTION

fmgetfontinfo returns information that pertains to the whole font in the space provided by the client in *info*. The information is described in the file *fmclient.h.* A note about some of the items:

printermatched means there is a printer widths file corresponding to this font.

matrix00–matrix11 are floats that provide information in points.

fixed_width means all the characters in the font are the same width.

xorig, yorig are the aggregate x-origin and y-origin of the font. *yorigin* is the positive distance between the baseline and the lowest descender. *xorigin* is the distance (positive or negative) from the current point to the start of imaging of the glyph.

xsize and ysize are the maximum sizes of the characters in the font, in pixels.

height is often the same as ysize, but some fonts lie to get more leading.

nglyphs is the index of the highest-numbered character and is not the total number of characters, but when allocating space for *fmgetwholemetrics,* use nglyphs as though it were the total number of characters. In other words, this is the highest index of a possibly sparse matrix.

The other items are self-explanatory.

SEE ALSO

fminit(3w), fmfindfont(3w), fmscalefont(3w).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetfontname – return the name associated with a font

SYNOPSIS

#include <fmclient.h>

long fmgetfontname(fh, slen, str);
fmfonthandle fh;
int slen;
char *str;

DESCRIPTION

fmgetfontname returns the name associated with a font in the space provided by the client in *str*. The length of the passed-in character array is passed in *slen; fmgetfontname* will not write more characters than specified by *slen*. If there is an error in locating the font specified by *fh*, a –1 is returned. If no name exists for the font, a -1 is returned, otherwise, the string's length is returned.

SEE ALSO

fminit(3w), fmfindfont(3W), fmscalefont(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetstrwidth – return the width of a string in pixels

SYNOPSIS

#include <fmclient.h>

long fmgetstrwidth(fh, str)
fmfonthandle fh;
char *str;

DESCRIPTION

fmgetstrwidth returns the number of pixels the string occupies in the x dimension. It uses the subpixel resolution provided in the glyph widths as it accumulates the width.

SEE ALSO

fminit(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmgetwholemetrics – return information about each of the characters in a font

SYNOPSIS

#include <fmclient.h>

long fmgetwholemetrics(fh, fi)
fmfonthandle fh;
fmglyphinfo *fi;

DESCRIPTION

fmgetwholemetrics gets the glyph (character) information associated with the font handle *fh* and writes it to the *fmglyphinfo* structures pointed to by the elements of *fi*. You should *calloc* enough space to contain *nglyphs*`*sizeof`(*fmglyphinfo*). *fmgetwholemetrics* fills only those structures of the array that have corresponding glyphs in the font file. It does not zero-out *fmglyphinfo* structures in excess of what it needs.

The returned function value of *fmgetwholemetrics* is 0 if successful. If *fmgetwholemetrics* cannot find the font referenced by *fh*, then the returned value is –1.

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W), fmgetfontinfo(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fminit – initialize the Font Manager

SYNOPSIS

#include <fmclient.h>

fminit()

DESCRIPTION

This routine should be called before calling any other routines in the Font Manager library. It sets the default page matrix that the font scaling and transformation routines use.

SEE ALSO

fmfindfont(3W), fmscalefont(3W), fmsetfont(3W), fmprstr(3W), fmfontpath(3W), fmsetpath(3W), fmenumerate(3W), fmgetfontinfo(3W), fmgetfontname(3W), fmgetstrwidth(3W), fmgetwholemetrics(3W), fmrotatepagematrix(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmmakefont – associate a transformation matrix with a font

SYNOPSIS

#include <fmclient.h>

fmfonthandle fmmakefont(fh, matrix)
fmfonthandle fh;
double matrix[3][2];

DESCRIPTION

fmmakefont concatenates the provided matrix to the matrix associated with this instance of the font, *returning a new handle.* When the font is imaged, this matrix is inspected to determine the proper scaling, shearing, rotation, or combination of these, for the imaging. This operator is more general than *fmscalefont,* which applies uniform scaling only.

SEE ALSO

fminit(3W), fmfontpath(3W), fmsetpath(3W), fmscalefont(3W), fmsetfont(3W), fmconcatpagematrix(3W), fmgetpagematrix(3W), fminitpagematrix(3W), fmrotatepagematrix(3W), fmscalepagematrix(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NOTES

Subsequent use of any of the routines discussed on the *fmmatrix* man page can cause a different transformation of the font to be either described (by *fmgetfontinfo* or *fmprstr*) or imaged. The page matrix is always concatenated with the font matrix before imaging a string or giving out information about a font.

NAME

fmmatrix: fminitpagematrix, fmsetpagematrix, fmgetpagematrix, fmscalepagematrix, fmrotatepagematrix, fmconcatpagematrix – font manager page matrix operations

SYNOPSIS

#include <fmclient.h>

void fminitpagematrix()

void fmsetpagematrix(m)
double m[3][2];

void fmgetpagematrix(m)
double m[3][2];

void fmscalepagematrix(x)
double x;

void fmrotatepagematrix(angle)
double angle;

void fmconcatpagematrix(m)
double m[3][2];

DESCRIPTION

The argument *m* points to a 3x2 matrix of doubles (floats), of which it is usually necessary to fill in only the first two rows when loading values.

The page matrix represents a mechanism for sizing and rotating fonts on the screen; it is not affected by nor does it affect the hardware matrix stack. The casual user need not call any of these routines to lookup and render characters of any size. Other, more mnemonic and complete routines are available for that. (However, *fmrotatepagematrix* is required to render fonts rotated.) The page matrix models a page, so, rotating the page matrix causes characters to be rotated when rendered. This differs from sizing characters, which can be done two ways. The first is to use *fmscalefont.* This associates a scaling factor to a given font, which, as in PostScript, begins as a 1-point-high set of characters. The second method is to call *fmscalepagematrix,* or to jam-load the page matrix with fmsetpagematrix. This would have the effect of magnifying any characters that are rendered by the scale factor of the page. The matrix associated with the font is concatenated with the page matrix, giving the final rendering size.

fminitpagematrix initializes the page matrix to an identity.

fmsetpagematrix loads the page matrix with the floating point values supplied in matrix *m*.

fmgetpagematrix returns the current page matrix in the matrix supplied by the client.

fmscalepagematrix uniformly scales the page matrix by *x*.

fmrotatepagematrix rotates the page matrix by the specified angle, where the angle increases up from the x axis.

fmconcatpagematrix concatenates matrix *m* supplied by the client with the page matrix with a post-multiply.

SEE ALSO

fminit(3W), fmgetfontinfo(3W), fmprstr(3W), fmscalefont(3W), fmsetfont(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmoutchar – render a single glyph.

SYNOPSIS

#include <fmclient.h>

long fmoutchar(fh, ch)
fmfonthandle fh;
unsigned char ch;

DESCRIPTION

fmoutchar renders a single glyph from the given font. If the glyph doesn't exist, it spaces forward the width of a space; if a space doesn't exist, it spaces forward the width of the font. The width used is returned.

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W), fmsetfont(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmprintermatch – toggle printer matching

SYNOPSIS

#include <fmclient.h>

void fmprintermatch(set)
int set;

DESCRIPTION

fmprintermatch(0) disables printer matching, *fmprintermatch(1)* enables printer matching. When a font is rendered (imaged), the state of this variable is inspected. If enabled, a printer widths file is sought that corresponds to the font. If the file exists, and the font has not yet been sized, a new font is created and inserted into the font handle that has widths that correspond to a laser printer's width scheme.

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W), fmsetfont(3W).
4Signt Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmprstr – render a string in the current font

SYNOPSIS

#include <fmclient.h>

long fmprstr(str)
char *str;

DESCRIPTION

fmprstr renders a string using the current font, as set by *fmsetfont.* It uses subpixel-positioning to assure that rounding errors do not cause aberrant spacing. It also renders rotated characters by inspecting and concatenating the page matrix with the font matrix, then rendering in the resulting coordinate system. If the string is null, or the font does not exist, *fmprstr* returns –1, otherwise, *fmprstr* returns the string length (rounded to the nearest pixel).

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W), fmsetfont(3W), fmrotatepagematrix(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmscalefont – scale a font face.

SYNOPSIS

#include <fmclient.h>

fmfonthandle fmscalefont(fh, scale)
fmfonthandle fh;
double scale;

DESCRIPTION

Findfont returns a handle to a 1-point-high font in the face specified. *fmscalefont* applies the provided scale factor to the matrix associated with this instance of the font, *returning a new handle.* Later, when the font is imaged, this value is used to determine the size of characters to use. Its default coordinate system is in points, so passing a scale of "12.0" creates a specification of a 12-point font. This is display-resolution independent. Its functionality parallels the scalefont operator of *PostScript.*

SEE ALSO

fminit(3W), fmfontpath(3W), fmsetpath(3W), fmscalefont(3W), fmsetfont(3W), fmconcatpagematrix(3W), fmgetpagematrix(3W), fminitpagematrix(3W), fmrotatepagematrix(3W), fmscalepagematrix(3W), fmsetpagematrix(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NOTES

Subsequent use of any of the routines discussed on the *fmmatrix* man page can cause a different transformation of the font to be either described (by *fmgetfontinfo* or *fmprstr*) or imaged. The page matrix is always concatenated with the font matrix before imaging a string or giving out information about a font.

NAME

fmsetcachelimit – set maximum cache size in quanta

SYNOPSIS

#include <fmclient.h>

void fmsetcachelimit(new_limit)
long new_limit;

DESCRIPTION

fmsetcachelimit accepts a small integer as an argument, multiplies it by FMCACHE_QUANTUM, and and uses the result to reset the upper limit of the font cache data space. If *new_limit* is less than one, *fmsetcachelimit* resets it to one before multiplying by FMCACHE_QUANTUM. If *new_limit* is greater than 100, *fmsetcachelimit* does nothing.

SEE ALSO

fmcachedisable(3W), fmcacheenable(3W), fmcachelimit(3W), fmgetcacheused(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NAME

fmsetfont – set the current font.

SYNOPSIS

#include <fmclient.h>

void fmsetfont(fh)
fmfonthandle fh;

DESCRIPTION

Findfont returns a handle to a 1-point-high font in the face specified. *fmscalefont* applies a scale factor to a face, and *fmsetfont* makes the font-face–size combination the current font. Any characters imaged with *fmprstr* or *fmoutchar* will be imaged using this font/size combination. Its functionality parallels the setfont operator of *PostScript.*

SEE ALSO

fminit(3W), fmfindfont(3W), fmscalefont(3W), fmprstr(3W), fmoutchar(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NOTES

Subsequent use of any of the routines discussed on the *fmmatrix* man page can cause a different transformation of the font to be either described (by *fmgetfontinfo* or *fmprstr*) or imaged. The page matrix is always concatenated with the font matrix before imaging a string or giving out information about a font.

NAME

fmsetpath – set the current font path.

SYNOPSIS

#include <fmclient.h>

void fmsetpath(path)
char *path;

DESCRIPTION

fmsetpath accepts a pointer to a string that describes the current search path for finding font files. It is a colon-separated list of directories that originate at the root. The default path is "/usr/lib/fmfonts".

SEE ALSO

fminit(3W), fmfontpath(3W).
4Sight Programmer's Guide, Section 1, "Using the GL/DGL Interfaces."

NeWS

NAME

bldfamily – build font family description

SYNOPSIS

bldfamily [–**d***dirname*] [–**o***outname*] [–v] *names*

DESCRIPTION

bldfamily scans a sets of NeWS font files and produces a NeWS font family file. A font family is a set of font files that are grouped together to provide a single PostScript font. In PostScript, a font has a name, like Times-Roman, and can be rendered in many different sizes. A NeWS font file is an instance of a PostScript font at a particular size. Font family files contain the information necessary for NeWS to pick the right bitmap font.

bldfamily scans directory *dirname* for files with extension ".font" or ".metrics" and where the leading non-digit characters of the filename match one of the *names* . The family file that is built will be written to *dirname/outname*.**ffam.**

If *outname* isn't specified, it defaults to the first of the *names* . If *dirname* isn't specified, it defaults to "$FONDIR" if defined, "." otherwise.

OPTIONS

–**d***dirname*	Specifies the directory to scan and put the **.ffam** file into.
–**o***outname*	Specifies the output font name
–v	Verbose – gives a more detailed description of what is going on.

EXAMPLE

```
% dumpfont -d /usr/newfonts -n Boston boston*.vfont
% bldfamily -d/usr/newfonts Boston
```

The first command calls *dumpfont* and converts all the vfont files whose names match "boston*.vfont" into NeWS format. It puts them into */usr/newfonts* and changes their font name to Boston (it would have defaulted to boston). The second command calls *bldfamily* and scans */usr/newfonts* for "Boston*.font" and builds a font family file for them, which will be called */usr/newfonts/Boston.ffam.*

SEE ALSO

dumpfont(1)

BUGS

The options and directory defaulting should be consistent with *dumpfont* . *bldfont* is clumsy to invoke.

NAME

cps – construct C to PostScript interface

SYNOPSIS

cps [**–c**] [**–D** *symbol*] [**–I** *filename*] [**–i**] [*PostScript file*]

DESCRIPTION

NeWS compiles a specification file containing C procedure names and PostScript code into a header file *filename.h* that can be included in C programs. Only one input file can be specified, and if the *filename.h* file has been previously created, a backup copy of this file will be generated in the form *filename.hBAK* before the new file is generated.

The convention is for the input specification file to end in *.cps*.

OPTIONS

–c
: Compiles a PostScript file for faster loading by NeWS, and is not used to generate a specification file for programs. For example, the following command line:

cps -c < input_file > output_file

will convert the *input_file* from the ascii form of the PostScript Language, to the compressed binary form. When read by NeWS, the *output_file* will execute exactly the same as *input_file*, except that it will be faster. The **–c** option **will not** work if the *input_file* uses constructs like currentfile readstring, which are often used with the image primitive.

–D *symbol*
: Defines symbols to be passed onto the C language preprocessor (*cpp*) which processes the input file.

–I *filename*
: Specifies include files. Passed on to the pre-processor.

–i
: Generates two specification files: one that contains only the C procedures and PostScript code that are user-defined, and one that contains other definitions required for the C-PostScript interface. For example, **ps_open_PostScript** and **ps_close_PostScript** would be defined in the second file. The second file references the user-defined procedures as **extern char.** The first file is of the form *filename.c*, and the second file is of the form *filename.h*. already exist.

This option is valuable for controlling the size of the CPS include files in multiple source files. The *filename.h* would only need to be included once. Each source file would only need to include its specific *filename.c* file generated by this option.

SEE ALSO

cpp(1).
4Sight Programmer's Guide, Section 2, "Programming in NeWS.

TRADEMARK

PostScript is a registered trademark of Adobe Systems, Inc.

NAME

dumpfont – dump font out in some other format

SYNOPSIS

dumpfont [**–a** | **–b** | **–v** | **–vf** | **–fm**] [**–c** *comment*] [**–d** *dirname*] [**–f** *n*]
[**–n** *fontname*] [**–S**] [**–s** *n*] [**–t**] [**–tv**] [**–ta**] *filenames*

DESCRIPTION

dumpfont reads in the set of named font files and dumps them out again according to the specified options, effectively converting the files from one font format to another. *dumpfont* is typically used to generate fonts for use with the NeWS window system.

There are five types of font files that *dumpfont* can read: Sun standard vfont format, Adobe ASCII bitmap format, Adobe ASCII metric format, NeWS font format, and CMU (Andrew) format. The format of the input font is determined automatically by inspecting the file. It can write fonts out in one of four formats: Adobe ASCII, NeWS, vfont, and Silicon Graphics Font Manager format. The default output format is NeWS.

OPTIONS

–a Selects Adobe ASCII output format. This is the format that you should use when transporting fonts from one machine architecture to another. The output file extension is ".afont".

–b Selects NeWS output format (the default). The output file extension is ".font". If the input file is an Adobe ASCII metrics file, the extension will be ".metrics".

–v Selects vfont output format. The output file extension is ".vfont".

–vf Selects vfont output format. The output file extension is ".vfont". Forces the characters to be fixed width.

–fm Selects SGI Font Manager output format. The output file extension is ".fm" or ".fw", for font data or font width data, respectively. *dumpfont* only writes this format, it does not read it.

−c *comment* Sets the *comment* field of the font. The Adobe ASCII and NeWS font formats support an internal comment that accompanies the font. This is usually used to contain copyright or history information. It is normally propagated automatically.

−d *dirname* Specifies the directory into which the font files will be written. If the FONTDIR environment variable is set, it is used as the default value. Otherwise, if the NEWSHOME environment variable is set, *$NEWSHOME/fonts* is used as the default value. Otherwise "." is used.

−f *n* Sets the maximum length of an output filename (excluding extension) to *n* . When writing NeWS format files, NeWS normally constructs the output filename from the name of the font and its scaling factors. Some systems cannot cope with long file names, so this option can be used to heuristically squeeze the name.

−n *name* Forces the output font name to be *name* . It is important not to confuse the name of the font with the name of the file that contains it. Some font formats (Adobe ASCII and NeWS) contain the name of the font internally. So given a 10-point Times-Roman font, its font name will be "Times-Roman", but its file name might be *TimRom10.font.*

−S Attempts to determine the size information of fonts by inspecting the bitmaps and applying some heuristics. This is useful when reading vfonts (particularly those intended for printers like the Versatec) that are missing or have incorrect size information.

−s *n* Sets the point size of the font to *n* . Overrides any internal size specification

−t Prints a short description of the fonts on standard output; a reformatted font file is not dumped.

–tv	Prints a move verbose description of the fonts on standard output; a reformatted font file is not dumped.
–ta	Prints a long description of the fonts on standard output; a reformatted font file is not dumped. You'll get every scrap of information.

SEE ALSO

bldfamily(1), vfont(5).
4Sight Programmer's Guide, Section 2, "Programming in NeWS."

DIAGNOSTICS

Bad flag: -C	Unknown command like option
Couldn't write ...	Error writing font file
f: not a valid font or not there.	The input file does not have a valid format, or the file cannot be found.

NAME

newshost – NeWS network security control.

SYNOPSIS

newshost add [*hosts*]
or **newshost remove** [*hosts*]
or **newshost show**

DESCRIPTION

Newshost is a shell command that manipulates the registry of hosts that are allowed to connect to the NeWS server. The identity of the NeWS server whose registry will be manipulated is determined by the NEWSSERVER environment variable. The variable **/NetSecurityWanted** (in the NeWS **systemdict** may be set to false to disable the security mechanism.

newshost add	adds the named hosts to the registry,
newshost remove	removes the named hosts from the registery,
newshost show	prints out a list of the hosts in the registry.

SEE ALSO

4Sight Programmer's Guide, Section 2, "Programming in NeWS."

NAME

psh – NeWS PostScript shell

SYNOPSIS

psh [**files**]

DESCRIPTION

psh opens a connection to the NeWS server and transmits the file arguments (or stdin if no files are specified) to it. Any output from NeWS is copied to stdout. The files should be PostScript programs for the NeWS server to execute.

A common use for *psh* is in creating applications written entirely in PostScript. First, type your PostScript program into a file. Then, type as its first line:

```
#! /usr/NeWS/bin/psh
```

If you now make the file executable (with *chmod*) you can invoke it by name from the shell, and UNIX will use */usr/NeWS/bin/psh* to execute it. *psh* will in turn send your program to the NeWS server.

SEE ALSO

sh(1), say(1)
4Sight Programmer's Guide, Section 2, "Programming in NeWS."

NAME

psio – NeWS buffered input/output package

SYNOPSIS

#include "psio.h"

PSFILE *psio_stdin;
PSFILE *psio_stdout;
PSFILE *psio_stderr;

DESCRIPTION

The functions described here constitute a user-level I/O buffering scheme for use when communicating with NeWS. This package is based on the standard I/O package that comes with Unix. The functions in this package are used in the same way as the similarly named functions in Standard I/O.

The in-line macros *psio_getc* and *psio_putc* handle characters quickly. The higher level routines *psio_read*, *psio_printf*, *psio_fprintf*, *psio_write* all use or act as if they use *psio_getc* and *psio_putc*; they can be freely intermixed.

A file with associated buffering is called a *stream*, and is declared to be a pointer to a defined type **PSFILE**. *psio_open* creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the *psio.h* include file and associated with the standard open files:

psio_stdin	standard input file
psio_stdout	standard output file
psio_stderr	standard error file

A constant **NULL** (0) designates a nonexistent pointer.

An integer constant **EOF** (–1) is returned upon end-of-file or error by most integer functions that deal with streams.

Any module that uses this package must include the header file of pertinent macro definitions, as follows:

```
#include "psio.h"
```

The functions and constants mentioned in here are declared in that header file and need no further declaration. The constants and the following 'functions' are implemented as macros; redeclaration of these names is perilous: *getc*, *putc*, *psio_eof*, *psio_error*, *psio_fileno*, and *psio_clearerr*.

SEE ALSO

open(2V), close(2), read(2V), write(2V), intro(3S), fclose(3S), ferror(3S), fopen(3S), fread(3S), getc(3S), printf(3S), putc(3S), ungetc(3S).
4Sight User's Guied, Section 2, "Programming in NeWS."

DIAGNOSTICS

The value **EOF** is returned uniformly to indicate that a **PSFILE** pointer has not been initialized with *psio_open*, input (output) has been attempted on an output (input) stream, or a **PSFILE** pointer designates corrupt or otherwise unintelligible **PSFILE** data.

LIST OF FUNCTIONS

Name	*Description*
psio_clearerr	stream status inquiries
psio_close	flush a stream
psio_eof	stream status inquiries
psio_error	stream status inquiries
psio_fdopen	open a stream
psio_flush	close or flush a stream
psio_fileno	stream status inquiries
psio_fprintf	formatted output conversion
psio_getc	get character or integer from stream
psio_open	open a stream
psio_read	buffered binary input/output
psio_printf	formatted output conversion
psio_putc	put character or word on a stream
psio_ungetc	push character back into input stream
psio_write	buffered binary input/output

NAME

psterm NeWS terminal emulator

SYNOPSIS

psterm [options] [command]

DESCRIPTION

psterm is a *termcap*-based terminal emulator program for NeWS. When invoked, it reads the */etc/termcap* entry for the terminal named by the **-t** option, or by the **TERM** environment variable, and arranges to emulate the behavior of that terminal. It forks an instance of *command* (or, by default, the program specified by the **SHELL** environment variable, or *csh* if this is undefined), routing keyboard input to the program and displaying its output.

psterm scales its font to make the number of rows and columns specified in the */etc/termcap* entry for the terminal it is emulating fit the size of its window. It also responds to most of the particular escape sequences that *termcap* defines for that terminal.

OPTIONS

−C route */dev/console* messages to this window, if supported by the operating system.

−f Bring up a reasonably-sized terminal in the lower-left corner of the screen (or in the location specified with the −xy option) instead of having the user define its size and location.

−w wait around after the *command* terminates.

−fl *frame label*
Use the specified string for the frame label.

−il *icon label*
Use the specified string for the icon label. The icon label normally defaults to the name of the host on which *psterm* is running.

−li *lines* specifies the height of the window in characters.

−co *columns*
specifies the width of the window in characters.

−xy *x y* specifies the location of the lower left hand corner of the window (in screen pixel coordinates).

−bg causes *psterm* to place itself in the background by disassociating itself from the parent process and the controlling terminal. If *psterm* is invoked with *rsh*(1), this option will cause the rsh command to complete immediately, rather than hang around until *psterm* exits.

–ls causes *psterm* to invoke the shell as a login shell. In addition, any specified *command* will be passed to the shell with a **–c** option, rather than being invoked directly, so that the shell can establish any environment variables that may be needed by the command. Further, if *psterm* is invoked via *rsh*(1), the host at the other end of the *rsh* socket will be used as the server, unless a NEWSSERVER environment variable is present.

–pm specifies that a *psterm* should enable *page mode*. When page mode is enabled and a command produces more lines of output that can fit on the screen at once, *psterm* will stop scrolling, hide the cursor, and wait until the user types a character before resuming output. When *psterm* is blocked with a screenfull of data, typing a carriage return or space will cause scrolling to proceed by one line or one screenful, respectively; any other character will cause the next screenfull to appear and be passed through as normal input. This mode can also be enabled or disabled interactively, using the *Page Mode* menu item.

SELECTION

Clicking the left mouse button over a character selects that character. Clicking it beyond the end of the line selects the newline at the end of that line. Clicking the middle mouse button over a character when a primary selection does not exist in that window selects that character. Clicking the middle mouse button over a character when a primary selection does exist in that window extends or shrinks the selection to that character.

Note that selections are made by **clicking**. Mouse tracking is not implemented yet.

The Copy key (F6) copies the *primary* selection to the *shelf*. The Paste key (F8) copies the contents of the *shelf* to the *insertion point*.

If you make a selection while holding down the Copy key, the selection will be a secondary selection. Subsequently letting go of the Copy key copies the *secondary* selection to the *shelf* and deselects the secondary selection.

Making a selection while holding down the Paste key also makes a secondary selection. It pastes the *primary* selection to the location of the secondary selection and deselects the secondary selection.

Copy and Paste of both primary and secondary selections work across separate invocations of *psterm.*

MENU ITEMS

Psterm adds two items to the top of the standard menu associated with the right hand mouse button. These items permit the page mode and automatic margin modes to be turned on and off. Menu items change according to the state of each mode. For example, if page mode is enabled, the menu item will indicate ''Page Mode Off''.

FILES

/etc/termcap to find the terminal description.

SEE ALSO

4Sight Programmer's Manual, Section 2, "Programming in NeWS."

BUGS

Emulating some terminal types works better than others, largely because there are incomplete */etc/termcap* entries for them.

A large number of *termcap* fields have yet to be implemented.

Page Mode gets easily confused.

NAME

psview – PostScript previewer for NeWS

SYNOPSIS

psview [*PostScript-file*]

DESCRIPTION

psview puts up a window and runs the user's PostScript code in it. *psview* uses a portion of the window that has the proper aspect ratio for a standard letter-size page in portrait orientation.

If *PostScript-file* is specified, the PostScript code is taken from that file. If no argument is given, or if a '-' is given as the argument, *psview* reads the PostScript program from standard input.

psview lets you flip through the pages. Page boundaries are determined by locating the %%Page: comments. *psview* provides a slider to move to any page, and a menu to go to the first, previous, next or last page. Clicking the left mouse button goes to the next page.

SEE ALSO

psh(1), say(1).
4Sight Programmer's Guide, Section 2, "Programming in NeWS."
PostScript Language Reference Manual.

TRADEMARK

PostScript is a registered trademark of Adobe Systems Inc.

BUGS

Assumes a syntactically valid PostScript file.

NAME

say – execute PostScript

SYNOPSIS

say [*options*] [*strings*]

DESCRIPTION

say connects to the NeWS server and displays the strings provided on the command line in a window. An option is provided to interpret the command line, or the standard input, as a PostScript program to be executed by the server.

say is used to implement some of the NeWS demo programs. This technique allows window applications to be shell scripts.

OPTIONS

–b*string*
: Use *string* as the title for the window.

–c Center the text in the window.

–p The command line contains a PostScript program rather than simply text strings.

–P The standard input contains a PostScript program, which is executed after the PostScript commands on the command line (if any).

–r Make the window round.

–s*nn* Use *nn* as the point size of the text.

–w Wait for the window to be destroyed. The default is for the window to vanish when execution of its PostScript program is finished.

–W Do not create a window for the PostScript to be executed in. This can be used to implement operations that do not require a window; for example toggling drag mode in the window manager, or running PostScript code that creates its own window.

–xxx,yyy
: The first *xxx,yyy* pair of numbers sets the X and Y coordinates of the window. If a second *–xxx,yyy* command line option is given, it sets the size of the window.

USAGE

Older programs that use *say* solely to send PostScript to the NeWS server (by specifying the **-P -w -W " "** options to *say*), should be converted to use *psh*(1).

SEE ALSO

psh(1).
4Sight Programmer's Guide, Section 2, "Programming in NeWS."
PostScript Language Reference Manual.

NAME

sc – spread sheet calculator

SYNOPSIS

sc [*file*]

DESCRIPTION

The spread sheet calculator *sc* is based on rectangular tables, in much the same style as Visicalc or Lotus 123. When it is invoked it presents you with an empty table organized as rows and columns of cells. Each cell may have a label string associated with it and an expression. The expression may be a constant or it may compute something based on other entries.

When *sc* is running, the screen is divided into four regions. The top line is for entering commands. The second line is for messages from *sc*. The third line and the first four columns show the row and column numbers. The rest of the screen forms a window looking at the table. The screen has two cursors: a cell cursor (indicated by a '<' on the screen) and a character cursor (indicated by the terminal's hardware cursor). The cell and character cursors are often the same. They will differ when a command is being typed on the top line.

Commands which use the terminal's control key such as ^N will work both when a command is being typed and when in normal mode.

The cursor control commands and the row, column commands can be prefixed by a numeric argument indicating how many times the command is to be executed. "^U" can be used before the number if the number is to be entered while a command is being typed into the command line.

Cursor control commands:

^N	Move the cell cursor to the next row.
^P	Move the cell cursor to the previous row.
^F	Move the cell cursor forward one column.
^B	Move the cell cursor backward one column.
^H	Backspace one character.

h, j, k, 1 Alternate cursor controls (left, down, up, right).

Arrow Keys
: The terminal's arrow keys provide another alternate set of cell cursor controls if they exist and are supported in the *termcap* entry. Some terminals have arrow keys which conflict with other control key codes. For example, a terminal could send ^H when the back arrow key is depressed. In these cases, the conflicting arrow key performs the same function as the key combination it mimics.

0 Move the cell cursor to column 0 of the current row.

$ Move the cell cursor to the last valid column in the current row.

^ Move the cell cursor to row 0 of the current column.

Move the cell cursor to the last valid row in the current column.

g Go to a cell. The program will prompt for the name of a cell. Enter a cell number such as "a0" or "ae122".

Cell entry and editing commands:

= Prompts for an expression which will be evaluated dynamically to produce a value for the cell pointed at by the cell cursor. This may be used in conjunction with ^V to make one entries value be dependent on anothers.

" Enter a label for the current cell.

< Enter a label that will be flushed left against the left edge of the cell.

> Enter a label that will be flushed right against the right edge of the cell.

x	Clears the current cell. You may prefix this command with a count of the number of cells on the current row to clear. Cells cleared with this command may be recalled with any of the variations of the pull command.
e	Edit the value associated with the current cell. This is identical to '=' except that the command line starts out containing the old value or expression associated with the cell.
E	Edit the string associated with the current cell. This is the same as either "leftstring", "rightstring", or "label", with the additional fact that the command line starts out with the old string.
m	Mark a cell to be used as the source for the copy command.
c	Copy the last marked cell to the current cell, updating the row and column references.
^T	Toggle cell display. The current cell's contents are displayed in line one when no command being entered or edited. ^T turns the display on or off.

File operations

G	Get a new database from a file.
P	Put the current database into a file.
W	Write a listing of the current database in a form that matches its appearance on the screen. This differs from the "put" command in that "put"s files are intended to be reloaded with "get", while "write" produces a file for people to look at.
T	Write a listing of the current database to a file, but put ":"s between each field. This is useful for tables that will be further formatted by the *tbl* preprocessor of *nroff*.

M Merges the database from the named file into the current database. Values, expressions and names defined in the named file are written into the current file, overwriting the existing entries at those locations.

Row and Column operations. Members of this class of commands can be used on either rows or columns. The second letter of the command is either a column designator (one of the characters c, j, k, ^N, ^p) or a row designator (one of r, l, h, ^B, ^F). Commands which move or copy cells also modify the variable references in affected cell expressions. Variable references may be frozen by using the "fixed" operator.

ar, ac Creates a new row (column) immediately following the current row (column). It is initialized to be a copy of the current one.

dr, dc Delete this row (column).

pr, pc, pm
Pull deleted rows (columns) back into the spread sheet. The last deleted set of cells is put back into the spread sheet at the current location. *pr* inserts enough rows to hold the data. *pc* inserts enough columns to hold the data. *pm* (merge) does not insert rows or columns. It overwrites the cells beginning at the current cursor location.

ir, ic Insert a new row (column) by moving the row (column) containing the cell cursor, and all following, down (right) one. The new position will be empty.

zr, zc Hide ("zap") the current row (column). This keeps a row or column from being displayed but keeps it in the data base.

vr, vc Removes expressions from the affected rows (columns), leaving only the values which were in the cells before the command was executed.

sr, sc Show hidden rows (columns). Type in a range of rows or columns to be revealed. The command default is the first range of rows or columns currently hidden.

f Sets the output format to be used for printing the numbers in each cell in the current column. Type in two numbers which will be the width in characters of a column and the number of digits which will follow the decimal point. Note that this command has only a column version and does have a second letter.

Region Operations: Region commands affect a rectangular region on the screen. All of the commands in this class start with a slash; the second letter of the command indicates which command to do. The program will prompt for needed paramters. Phrases surrounded by square brackets in the prompt are informational only and may be erased with the backspace key.

/x Clear a region. Cells cleared with this command may be recalled via any of the pull row or column commands.

/c Copy a region to the area starting at the current cell.

/f Fill a region with constant values. The start and increment numbers may be positive or negative.

Miscellaneous commands:

q Exit from *sc*. If you were editing a file, and you modified it, then *sc* will ask about saving before exiting. If you aren't editing a file and haven't saved the data you entered, you will get a chance to save the data before you exit.

^C Alternate exit command.

? Types a brief helpful message.

^G or ESC
Abort entry of the current command.

^R or ^L Redraw the screen.

^V	Types, in the command line, the name of the cell referenced by the cell cursor. This is used when typing in expressions which refer to entries in the table.
^E	Types, in the command line, the expression of the cell referenced by the cell cursor.
^A	Types, in the command line, the value of the cell referenced by the cell cursor.

Expressions that are used with the '=' and 'e' commands have a fairly conventional syntax. Terms may be variable names (from the ^V command), parenthesised expressions, negated terms, and constants. Rectangular regions of the screen may be operated upon with '@' functions such as sum (@sum), average (@avg) and product (@prod). Terms may be combined using many binary operators. Their precedences (from highest to lowest) are: ^; *,/; +,-; <,=,>,<=,>=; &; |; ?.

e+e	Addition.
e-e	Subtraction.
e*e	Multiplication.
e/e	Division.
e^e	Exponentiation.
@sum(v:v)	Sum all valid (nonblank) entries in the region whose two corners are defined by the two variable (cell) names given.
@avg(v:v)	Average all valid (nonblank) entries in the region whose two corners are defined by the two variable (cell) names given.

@prod(v:v)	Multiply together all valid (nonblank) entries in the region whose two corners are defined by the two variable (cell) names given.
e?e:e	Conditional: If the first expression is true then the value of the second is returned, otherwise the value of the third is.
<,=,>,<=,>=	Relationals: true iff the indicated relation holds.
&,|	Boolean connectives.
fixed	To make a variable not change automatically when a cell moves, put the word ''fixed'' in front of the reference. I.e. B1*fixed C3

Assorted math functions. Most of these are standard system functions more fully described in *math*(3). All of them operate on floating point numbers (doubles); the trig functions operate with angles in radians.

@exp(expr)	Returns exponential function of <expr>.
@ln(expr)	Returns the natural logarithm of <expr>.
@log(expr)	Returns the base 10 logarithm of <expr>.
@pow(expr1,expr2)	Returns <expr1> raised to the power of <expr2>.
@floor(expr)	Returns returns the largest integer not greater than <expr>.
@ceil(expr)	Returns the smallest integer not less than <expr>.
@hypot(x,y)	Returns SQRT(x*x+y*y), taking precautions against unwarranted overflows.

@fabs(expr)	Returns the absolute value |expr|.
@sin(expr), @cos(expr), @tan(expr)	Return trigonometric functions of radian arguments. The magnitude of the arguments are not checked to assure meaningful results.
@asin(expr)	Returns the arc sin in the range -pi/2 to pi/2
@acos(expr)	Returns the arc cosine in the range 0 to pi.
@atan(expr)	Returns the arc tangent of <expr> in the range -pi/2 to pi/2.
@dtr(expr)	Converts <expr> in degrees to radians.
@rtd(expr)	Converts <expr> in radians to degrees.
pi	A constant quite close to pi.
@max(expr1,expr2)	Returns the largest value of the two expressions.
@min(expr1,expr2)	Returns the smallest value of the two expressions.
@gamma(expr1)	Returns the natural log of the gamma function.

SEE ALSO

bc(1), dc(1).
4Sight User's Guide, Section 2, "Programming in NeWS."

BUGS

At most 200 rows and 40 columns.

NAME

setnewshost – generate a string for the NEWSSERVER environment variable

SYNOPSIS

setnewshost *hostname*

DESCRIPTION

setnewshost generates and prints the proper value of the NEWSSERVER environment variable for the given *hostname*. If NEWSSERVER is set then NeWS clients will attempt to connect to the server it points to rather than the local host.

The format of the NEWSSERVER environment variable is as follows:

decimal-address **.** *port#* ; *hostname*

For example, if the host called ''paper'' has address 192.98.34.118, the NEWSSERVER variable should be set to ''3227656822.2000;paper'' so that NeWS clients will connect to the NeWS server on ''paper''. *setnewshost* simply calculates this string and sends it to standard output. This is not its most convenient form, however. C-shell users can define the following alias:

```
alias snh 'setenv NEWSSERVER `setnewshost \!*`'
```

and System V Bourne Shell users can define the following function:

```
snh () {
   NEWSSERVER=`setnewshost $*`
   export NEWSSERVER
}
```

Both forms let you simply type '**snh** *hostname*' to set the NEWSSERVER environment variable automatically.

SEE ALSO

psh(1).
4Sight Programmer's Guide, Section 2, "Programming in NeWS."

BUGS

The host table entry must have exactly the following format: **a.b.c.d**<*tab*>**hostname.**

If you use the **snh** alias or shell function, and the hostname you give is unknown, or you give too many or too few arguments, the NEWSSERVER variable will be trashed.

NAME

winicons – stowed window image mechanism

DESCRIPTION

When an active window is stowed by the user, an RGB image file may be used to paint the canvas of the stowed window icon. A window icon file must contain the suffix *.icon.*

Window icons are assigned to stowed windows by matching the name that appears in the program's call to the Graphics Library subroutine, `winopen`. Thus, an icon for the GL program *cedit* would have this name:

```
cedit.icon
```

WINICON SEARCH PATH

A directory of default window icons exists in *$NEWSHOME/icons.* You may add or customize window icons by placing them in *$HOME/.4sight/icons.* To find the appropriate window icon for a stowed window, 4Sight first searches *$HOME/.4sight/icons* for a name match. If it is unsuccessful, it seaches the default window icon directory (*$NEWSHOME/icons*). If this is unsuccessful, it uses the prededined icon *$NEWSHOME/icons/default.icon.* If this icon is missing for some reason, 4Sight draws the icon without an image.

CREATING A WINICON FILE

A window icon can be created from any image that can be displayed on the IRIS screen (provided that the tools described below are accessible when the image is displayed).

The following passage describes one possible way to create an icon image file. First, display the image you wish to use with the image tools *ipaste* or *showci,* or simply open a window containing a program from which you want to take an image (make sure that the image is still). Then invoke the image tool *icut* from the command line. *icut* takes a filename as an argument; the image cut from the screen is written to that file.

```
icut foo
```

Place the small rectagular *icut* window away from the image you wish to cut. Place the mouse cursor inside the icut window and hold down the <shift> key. While holding down the key, move the mouse cursor to the upper left corner of the area you wish to cut. Hold down the left mouse button while you move to the lower right corner of the area you wish to cut (the area is not shown on the screen), and when you are ready, let go of the left button. The image is then written to the file.

Note: The image you cut must be scaled to fit the stowed window canvas, so you should attempt to cut an area of the same general shape as the icon. The ratio of window icon width to height is 64:50.

You can use the file obtained with *icut* as a window icon file, and 4Sight will scale and dither it on the fly. However, to increase efficiency and image quality, you may want to scale it yourself. To do so, first run the image tool *istat* on the *icut* file:

```
istat foo
```

istat gives a readout of various image statistics; the important ones for scaling are the first two values, the image width (*xsize*) and image height (*ysize*). The dimension of stowed windows is 50 NeWS (4Sight) points high by 64 NeWS points wide. The ratio of screen pixels to points is 4:3; this yields the following scaling factors:

$$xscale = 85.33/(xsize)$$
$$yscale = 66.67/(ysize)$$

To scale the image, use the image tool *izoom. izoom* takes in input file, an output file, and width and height scaling factors as arguments. To scale an image file *foo* whose dimensions are 620 pixels wide by 500 pixels high and make it into a console window icon, you would type the following on the command line:

```
izoom foo $HOME/.4sight/icons/console.icon .137 .133
```

NOTES

Some programs do not use this mechanism to draw their window icons; specifically, those that draw their own icons rather than let 4Sight do it for them. Certain NeWS programs, such as the Calculator draw their own icons in PostScript; Graphics Library programs using `iconsize` to draw their icons use Graphics Library commands to do so. Both of these methods override the window icon mechanism described here.

SEE ALSO

ipaste(1G), iconsize(3G)

Silicon Graphics, Inc.

Date ______________________

Your name ______________________

Title ______________________

Department ______________________

Company ______________________

Address ______________________

____

____

Phone ______________________

COMMENTS

Manual title and version ______________________

____

Please list any errors, inaccuracies, or omissions you have found in this manual

____

____

____

____

Please list any suggestions you may have for improving this manual

____

____

____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Silicon Graphics, Inc.
Attention: Technical Publications
2011 Stierlin Road
Mountain View, CA 94043-1321