Chapter 10

Framebuffers and Drawing Modes

This chapter describes modes used for accessing different "layers" of your graphics scene. The subroutines described in this chapter let you draw an image on top of the standard pixel contents (*overlay*) or underneath (*underlay*). Personal IRIS and IRIS Indigo owners with the minimum bitplane configuration (eight bitplanes) can skip the sections dealing with overlay and underlay bitplanes, because these bitplanes are not present on these systems.

- Section 10.1, "Framebuffers," describes the framebuffers on the IRIS workstation.
- Section 10.2, "Drawing Modes," describes the modes used for drawing.
- Section 10.3, "Writemasks," tells you how to use masks to draw selectively into designated layers.
- Section 10.4, "Configuring Overlay and Underlay Bitplanes," tells you how to designate drawing layers for multilayer drawings.
- Section 10.5, "Cursor Techniques," tells you how to create and use cursors.

10.1 Framebuffers

Pixel data is stored in a special memory called the *framebuffer*. In its default configuration, the framebuffer on IRIS workstations is divided into four partitions that the GL can address as if they were separate framebuffers or layers of bits called *bitplanes*. These four bitplanes are called: *normal*, *overlay*, *underlay*, and *pop-up*. Setting a *drawing mode* selects one of these bitplanes for access.

The exact number of bits varies from system to system and is also dependent on the drawing mode. Use `getgdesc()` with the appropriate `GD_BITS` parameter to query the system for the bit configuration in each mode.

RealityEngine systems have additional framebuffers, a multisample buffer for antialiasing, and left and right buffers for stereo applications.

10.1.1 Normal Framebuffer

Usually, drawing is done in the standard (normal) color framebuffer. Data in the framebuffer is interpreted either as RGB and, optionally, Alpha, data for RGB mode, or as indices into a color map for color map mode. A single sample of the color is written to the framebuffer to represent 1 pixel.

The color represented by a pixel in the framebuffer is not necessarily the color drawn on the screen. Colors from other framebuffers can appear on top of or underneath this pixel, or colors can be blended with the pixel to achieve interesting effects such as transparency. For example, a cursor moving over a pixel obscures the pixel's color while the cursor is displayed in that location. The original color of the pixel is displayed once again after the cursor passes. Similarly, when a pop-up menu is drawn over a window, the underlying colors are temporarily obscured, but they reappear when the pop-up menu disappears. These effects are accomplished by drawing into other parts of the framebuffer such as the overlay, underlay, or pop-up planes.

On RealityEngine systems, a *multisample* buffer coexists with the normal framebuffer that provides single-pass multisample antialiasing, which is described in Chapter 15. Multisampling can be used only when the draw mode is `NORMALDRAW`, and only when you have configured the multisample buffer.

RealityEngine systems also feature a flexible framebuffer configuration that allows you to specify how to partition the framebuffer and how to allocate bits within the various parts of the framebuffer. You can configure the framebuffer to choose the number of subsamples for multisample antialiasing and you can also allocate framebuffer bits for z-buffer and stencil operations.

10.1.2 Overlay Framebuffer

Overlay planes are useful for creating menus, construction lines, rubber-banding lines, and so on. Overlay bitplanes supply additional bits of information at each pixel. You can configure the system to have from 0 up to a system-dependent maximum number of bitplanes. Whenever all the overlay bitplanes contain 0 at a pixel, the color of the pixel from the standard color bitplanes is presented on the screen. If the value stored in the overlay planes is not 0, the overlay value is looked up in a separate color table, and that color is presented instead.

10.1.3 Underlay Framebuffer

Underlay planes are useful for background grids that appear where nothing else is drawn, such as a reference grid for a sketching application. Underlay bitplanes are similar in concept, in that there are extra bits for each pixel, but their values are normally ignored unless the color in the standard bitplanes is 0. In that case, the underlay color is looked up in a color map and is presented. Thus, the underlay color shows up only if there is "nothing" (the pixel value = 0) in the standard bitplanes. With two underlay bitplanes, there are four possible underlay colors.

Use `drawmode()`, described in Section 10.2, "Drawing Modes," to enter overlay or underlay mode and to return to normal drawing mode—drawing into the standard bitplanes.

The system actually has several physical bitplanes that can be used for either overlay or underlay. Two of the available bitplanes are normally reserved for window manager use; you can allocate the others among overlay bitplanes, underlay bitplanes, or neither. See Section 10.4, "Configuring Overlay and Underlay Bitplanes."

10.1.4 Pop-up Framebuffer

The two bitplanes normally reserved by the window manager for pop-up menus are accessible (either by themselves using a special drawing mode, or by allocating all the available overlay and underlay bitplanes). You must be careful, however, not to conflict with the window manager's use for them, so using the reserved bitplanes is not recommended.

10.1.5 Left and Right Stereo Framebuffers

This section describes a feature that is available only on RealityEngine systems, so you may want to skip to Section 10.2, “Drawing Modes,” if you do not have one of these systems.

On RealityEngine systems, the normal framebuffer can be configured for stereoscopic viewing with the `stereobuffer()` command.

When `stereobuffer()` is used in conjunction with `singlebuffer()`, two color buffers, left and right, are allocated. When `stereobuffer()` is used in conjunction with `doublebuffer()`, four color buffers, front-left, front-right, back-left, and back-right, are allocated.

`stereobuffer()` does not take effect until `gconfig()` is called. The default mode is monoscopic viewing. If neither `monobuffer()` nor `stereobuffer()` is called, `gconfig()` defaults to monoscopic buffer mode.

`stereobuffer()` configures the framebuffer to store left and right images, but it does not position those images as required for stereo viewing. Use `setmonitor()` to select a stereo video format to display the stereo images correctly. See Chapter 5 for the `setmonitor()` parameters. When `setmonitor()` is not configured for stereo display, only the left buffer is displayed.

In singlebuffer mode, `leftbuffer()` controls whether drawing is enabled in the left buffer, and `rightbuffer()` controls whether drawing is enabled in the right buffer.

In doublebuffer mode, the front-left buffer is enabled for drawing if both `frontbuffer()` and `leftbuffer()` are true, and the back-left buffer is enabled for drawing if both `backbuffer()` and `leftbuffer()` are true. Likewise, the front-right buffer is enabled for drawing if both `frontbuffer()` and `rightbuffer()` are true, and the back-right buffer is enabled if both `backbuffer()` and `rightbuffer()` are true.

`leftbuffer()` and `rightbuffer()` should be called only when the draw mode is `NORMALDRAW`, and they are ignored when the normal buffer is not configured for stereo buffering.

TRUE is the default for `leftbuffer()` and FALSE is the default for `rightbuffer()`. After `gconfig()` is called, the left buffer is enabled and the right buffer is disabled.

10.2 Drawing Modes

The drawing mode specifies which of the four layers, *normal*, *overlay*, *underlay*, or *pop-up*, is the intended destination for the bits produced by subsequent drawing and mode commands. Calls to `color()`, `getcolor()`, `getwritemask()`, `writemask()`, `mapcolor()`, and `getmcolor()` are affected by the current drawing mode.

Rather than introduce a new set of subroutines for operating on the different layers, the color map subroutines are used to affect the overlay and underlay bitplanes if the system is in overlay or underlay mode.

For example, in overlay mode, all drawing routines draw into the overlay bitplanes rather than into the standard bitplanes. In overlay mode, `color()` sets the overlay color; `getcolor()` gets the current overlay color; `mapcolor()` affects entries in the overlay map, and `getmcolor()` reads those entries. The routines are similarly redefined for underlay mode.

Some system resources are shared among the bitplanes, while in other cases the system maintains individual resources for each one. The pop-up, overlay, normal, and underlay planes maintain a separate version of each of the following modes, which are modified and read back, based on the current drawing mode:

`backbuffer`
`cmode`
`color` or `RGBcolor`
`doublebuffer`
`frontbuffer`
`mapcolor` (a complete separate color map)
`readsource`
`RGBmode`
`singlebuffer`
`writemask` or `RGBwritemask`

Other modes affect only the operation of the normal framebuffer. You can modify these modes only while the normal framebuffer is selected:

```
acsize
blink
cyclemap
multimap
onemap
setmap
stencil
stensize
swritemask
zbuffer
zdraw
zfunction
zsource
zwritemask
```

All other modes are shared, including matrices, viewports, graphics and character positions, lighting, and many primitive rendering options.

There is a special bitplane area reserved for cursor images. See Section 10.5, "Cursor Techniques." In cursor mode, only `mapcolor()` and `getmcolor()` perform a function; `color()`, `getcolor()`, `writemask()`, and `getwritemask()` are ignored.

drawmode

`drawmode()` sets the current drawing mode; *mode* defines the drawing mode:

```
void drawmode(long mode)
```

Drawing modes are:

`UNDERDRAW`	Sets operations for the underlay planes.
`NORMALDRAW`	Sets operations for the normal color index or RGB planes; also sets *z* bitplanes.
`OVERDRAW`	Sets operations for the overlay planes.
`PUPDRAW`	Sets operations for the pop-up menu planes (this drawing mode is maintained for compatibility only and is not recommended).
`CURSORDRAW`	Sets operations for the cursor planes.

The default drawing mode is NORMALDRAW mode, which remains set unless you change it explicitly.

getdrawmode

`getdrawmode()` returns the current drawing mode specified by `drawmode()`:

```
long getdrawmode(void)
```

Each drawing mode has its own color and *writemask*. See Section 10.3, "Writemasks," for information about writemasks. By default, the writemask enables all planes, and the color is not defined. As you switch from one drawing mode to another, the current color and writemask are saved, and the previously saved color and writemask for the new mode are restored. For example, if you are in NORMALDRAW, then switch to OVERDRAW, and then switch back to NORMALDRAW, the color and writemask that were active before you switched to OVERDRAW are automatically restored.

You can use Gouraud shading, that is, `shademodel(GOURAUD)` in NORMALDRAW mode. On the Personal IRIS, when you draw polygons in the overlay and underlay bitplanes, or pop-up menus, the shading model is automatically set to FLAT.

Many routines that affect the operation of the standard bitplanes should not be used while in overlay or underlay drawing mode. They include `doublebuffer()` (on all except VGX, SkyWriter, and RealityEngine systems), `RGBmode()`, `zbuffer()`, and `multimap()`. VGX, SkyWriter, and RealityEngine systems support double-buffered underlay and overlay.

10.3 Writemasks

In cases when the system uses color maps (the standard bitplanes in color map mode, and the overlay and underlay bitplanes), a *writemask* is available that can selectively limit drawing into the bitplanes. A writemask determines whether or not a new value is stored in each bitplane. A one in the writemask allows the system to store a new value in the corresponding bitplane; a zero prevents a new value from being written, so the bitplane retains its current color.

By default, the writemask is set up so that there are no drawing restrictions, but it is sometimes useful to limit the effects of the drawing routines. The two most common cases are to provide the equivalent of extra overlay bitplanes and to display a layered scene where the contents of the layers are independent of one another. In previous systems, overlay and underlay modes were not available; consequently, writemasks had a more significant function.

10.3.1 How Writemasks Work

Figure 10-1 shows how a writemask works. In this example, the values in the first and second bits (*b1* and *b2*) do not change because their corresponding positions in the writemask are zero. All the other values change because they have ones in their corresponding positions in the writemask.

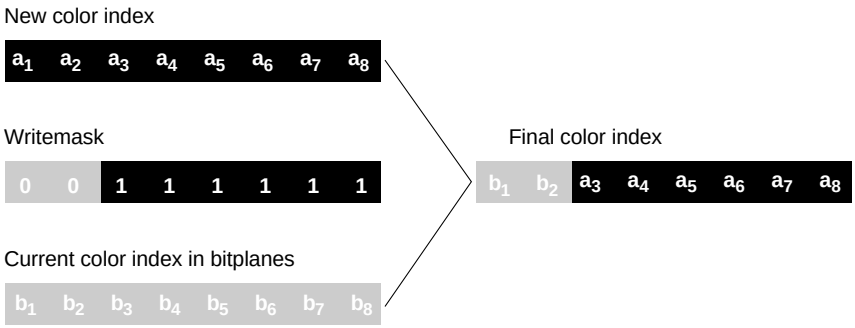


Figure 10-1 Writemask

The writemask is described here in terms of the standard drawing bitplanes, but it works exactly the same way if the system is in overlay or underlay mode. This discussion assumes that only 8 of the 12 bitplanes are used, although the discussion applies equally well to different numbers.

With 8 bitplanes, the color is a number from 0 to 255, which can be represented by 8 binary bits. For example, color 68 is 01000100 in binary notation. Without writemask controls, if the color is set to 68, every drawing subroutine puts 01000100 into the 8 bitplanes of the affected pixels.

A writemask restricts what is written to the bitplanes. In the example above, if the writemask is 15 (bits= 00001111), only the bottom (right-most) 4 bits of the

color are written into the bitplanes (1 enables writing to the bitplane; 0 disables writing to the bitplane). If the color is 68, any pixels hit by a drawing subroutine while the writemask is enabled contain ABCD0100, where ABCD are the 4 bits that were previously there. The zeros in the writemask prevent those bits from writing. The default writemask is entirely ones, so there is no restriction.

10.3.2 Writemask Subroutines

This section describes the subroutines you use to work with writemasks.

writemask

`writemask()` grants write permission to available bitplanes in color map mode:

```
void writemask(Colorindex wtm)
```

The writemask prevents writing into (protects) bitplanes in the current drawing mode that are reserved for special uses. *wtm* is a mask with 1 bit available per bitplane. Wherever there are ones in the writemask, the corresponding bits in the color index are written into the bitplanes. Zeros in the writemask mark bitplanes as read-only. These bitplanes are unchanged, regardless of the bits in the color.

If the drawing mode is `NORMALDRAW`, `writemask()` affects the standard bitplanes; if it is `OVERDRAW`, the overlay bitplanes; if it is `UNDERDRAW`, the underlay bitplanes. It is important to understand that although `writemask()` allows you to protect certain bits from being overwritten, all the bits stored at any pixel are still taken as a single integer or color index value (see the *circles.c* sample program).

RGBwritemask

`RGBwritemask()` is the same as `writemask()`, except that it functions in RGB mode:

```
void RGBwritemask(short red, short green, short blue)
```

The arguments *red*, *green*, and *blue* are masks for each of the three sets of bitplanes. In the same way that writemasks affect drawing in bitplanes in

NORMALDRAW color map mode, separate red, green, and blue masks can be applied in NORMALDRAW RGB mode.

wmpack

wmpack() is the same as RGBwritemask(), except it that it accepts a single packed argument, rather than three separate masks:

```
wmpack(unsigned long pack)
```

Bits 0 through 7 specify the red mask, 8 through 15 the green mask, 16 through 23 the blue mask, and 24 through 31 the alpha mask. For example, wmpack(0xff804020) has the same effect as RGBwritemask(0x20,0x40,0x80).

getwritemask

In color map mode, getwritemask() returns the current writemask of the current drawing mode:

```
long getwritemask(void)
```

The return value is an integer with up to 12 significant bits, one for each available bitplane.

gRGBmask

gRGBmask() returns the current RGB writemask as three 8-bit masks:

```
void gRGBmask(short *redm, *greenm, *bluem)
```

gRGBmask() places masks in the low order 8-bits of the locations *redm*, *greenm*, and *bluem* address. The system must be in RGB mode when this routine executes.

10.3.3 Sample Writemask Programs

Two sample programs that use writemasks follow.

The first sample program, *circles.c*, draws overlapping circles. Because of the writemask, the overlapping colors form their compound color—that is, where

the red and green circles overlap, the shared area is yellow; where the red and blue circles overlap, the shared area is magenta, and so on.

```
#include <gl/gl.h>
main()
{
   prefsize(400, 400);
   winopen("circles");
   color(BLACK);
   clear();
   writemask(RED);
   color(RED);
   circfi(150, 250, 100);
   writemask(GREEN);
   color(GREEN);
   circfi(250, 250, 100);
   writemask(BLUE);
   color(BLUE);
   circfi(200, 150, 100);
   sleep(10);
   gexit();
   return 0;
}
```

As a more involved example, suppose you want to draw two completely independent electronic circuits on the screen: power and ground. You want the circuit to be drawn on a white background, with the power traces in blue, the ground traces in black, and short circuits (power touching ground) in red.

Initialize the program as follows:

```
#define   BACKGROUND          0  /*=00*/
#define   POWER               1  /*=01*/
#define   GROUND              2  /*=10*/
#define   SHORT               3  /*=11*/
mapcolor(0, 255, 255, 255);      /*white*/
mapcolor(1, 0, 0, 255);          /*blue*/
mapcolor(2, 0, 0, 0);            /*black*/
mapcolor(3, 255, 0, 0);          /*red*/
```

Draw all the power circuitry into bitplane 1 and the ground circuitry into bitplane 2. Where both power and ground appear, there is a 1 in both bitplanes, making color 3.

To clear the window before drawing:

```
writemask(3);
color(BACKGROUND);
clear();
```

To draw power circuitry without affecting ground circuitry:

```
writemask(1);
color(1);
<drawing subroutines>
```

To draw ground circuitry without affecting power circuitry:

```
writemask(2);
color(2);
<drawing subroutines>
writemask(1);
color(0);
clear();
```

To erase all ground circuitry:

```
writemask(2);
color(0);
clear();
```

The complete *circuit.c* sample program follows. The user interface consists of the keys **P** (draw power rectangles), **G** (draw ground rectangles), **C** (clear the window), and **Q** (quit). To draw a rectangle, press the left mouse button at one corner, hold it down, slide the mouse to the other corner, and release it.

When you use the program, be sure to exit by typing **Q**—this resets the four lowest entries in the color map (which are used by all the windows) back to the default values. If you forget, your text will be white against a white background, and hence a bit tough to read. If this happens, type a couple of carriage returns, followed by `gclear()` and another carriage return. The program `gclear()` resets the color map back to the default.

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

#define R               0
#define G               1
#define B               2
#define RGB             3
#define BACKGROUND 0x0              /* = 00 */
```

```
#define POWER 0x1                    /* = 01 */
#define GROUND 0x2                   /* = 10 */
#define SHORT 0x3                    /* = 11 */

void powerrect(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;
{
    writemask(0x1);
    color(POWER);
    sboxfi(x1, y1, x2, y2);
}

void groundrect(x1, y1, x2, y2)
Icoord x1, y1, x2, y2;
{
    writemask(0x2);
    color(GROUND);
    sboxfi(x1, y1, x2, y2);
}

void clearcircuit()
{
    writemask(0x3);
    color(BACKGROUND);
    clear();
}

main()
{
    int i, drawtype;
    Device dev;
    short val;
    short x1, y1, x2, y2;
    long xorg, yorg;
    Boolean run;
    short saved[SHORT+1][RGB];
    prefsize(400, 400);
    winopen("circuit");
    color(BLACK);
    clear();
    qdevice(PKEY);                   /* draw power rectangles */
    qdevice(GKEY);                   /* draw ground rectangles */
    qdevice(CKEY);                   /* clear screen */
    qdevice(ESCKEY);                 /* quit */
    qdevice(WINQUIT);                /* quit from window manager */
    qdevice(LEFTMOUSE);              /* mark rectangle corners */
```

```
    tie(LEFTMOUSE, MOUSEX, MOUSEY);
    getorigin(&xorg, &yorg);

/* save existing color map */
    for (i = BACKGROUND; i <= SHORT; i++)
        getmcolor(i, &saved[i][R], &saved[i][G], &saved[i][B]);

/* load new color map */
    mapcolor(BACKGROUND, 0, 0, 0);              /* black */
    mapcolor(POWER, 0, 0, 255);                 /* blue */
    mapcolor(GROUND, 255, 255, 255);            /* white */
    mapcolor(SHORT, 255, 0, 0);                 /* red */
    drawtype = GROUND;
    run = TRUE;
    while (run) {
        dev = qread(&val);
        if (dev == WINQUIT)
            run = FALSE;
        else if (dev == LEFTMOUSE) {            /* downclick */
            qread(&x1);
            qread(&y1);
            qread(&val);                        /* upclick */
            qread(&x2);
            qread(&y2);
            if (drawtype == POWER)
                powerrect(x1 - xorg, y1 - yorg, x2 - xorg, y2 - yorg);
            else
                groundrect(x1 - xorg, y1 - yorg, x2 - xorg, y2 - yorg);
        }
        else if (val == 0) {/* on upstroke only */
            switch (dev) {
            case PKEY:
                drawtype = POWER;
                break;
            case GKEY:
                drawtype = GROUND;
                break;
            case CKEY:
                clearcircuit();
                break;
            case ESCKEY:
                run = FALSE;
                break;
            }
        }
    }
```

```
    /* restore default color map */
    for (i = BACKGROUND; i <= SHORT; i++)
        mapcolor(i, saved[i][R], saved[i][G], saved[i][B]);
    gexit();
    return 0;
}
```

10.4 Configuring Overlay and Underlay Bitplanes

To set the number of user-defined bitplanes to use for overlay color or underlay color, call `overlay()` or `underlay()`, respectively.

Not all systems support overlay and underlay user-defined bitplanes simultaneously. Personal IRIS and IRIS-4D / G / GT / GTX systems support only one or the other at any one time. Call `gconfig()` after `overlay()` or `underlay()` to activate their settings `overlay()/underlay()` takes effect only after `gconfig()` is called, which is when all bitplane requests are resolved.

overlay

`overlay()` sets the number of user-defined bitplanes used for overlay colors:

```
void overlay(long planes)
```

underlay

`underlay()` sets the number of user-defined bitplanes used for underlay color:

```
void underlay(long planes)
```

`underlay()` is the same as `overlay()` except it affects the `underlay()` colors. The default value is 0.

planes is the number of bitplanes to use for the overlay / underlay color, which depends on the system type. The overlay / underlay framebuffer contains two bitplanes in single buffer, color map mode. Use `overlay()/underlay()` to change the number of bitplanes allocated for overlay / underlay mode. Use `drawmode()` to specify the overlay / underlay planes as the destination for drawing mode changes.

Table 10-1 shows the system type and the number of overlay / underlay planes.

System	Overlay/Underlay Planes
Personal IRIS	0 or 2 single buffer, color-map mode overlay / underlay bitplanes. (There are no overlay / underlay bitplanes in the minimum configuration of the Personal IRIS.)
IRIS-4D / G IRIS-4D / GT IRIS-4D / GTX	0, 2, or 4 single buffer, color-map mode overlay / underlay bitplanes. (Use of 4 is discouraged, because of interference with the window manager pop-up bitplanes.)
IRIS-4D / VGX, SkyWriter	0, 2, 4, or 8 single or double buffer color map mode overlay / underlay bitplanes. The 4- and 8-bitplane configurations use the alpha bitplanes, which are then unavailable for use in `NORMALDRAW` mode. Furthermore, your system must have the alpha bitplane option for you to use 4 or 8 overlay / underlay bitplanes with an IRIS-4D / VGX or SkyWriter system.
RealityEngine	0,2,4, or 8 single or double buffer overlay / underlay bitplanes. Does not steal alpha planes.
IRIS Indigo	0

Table 10-1 Overlay and Underlay Bitplane Configurations

On models that cannot support simultaneous overlay and underlay, setting `underlay()` to 2 forces `overlay()` to 0 and vice-versa. If `underlay()` is 2, there are four available colors that `mapcolor()` can define. This is one more than the available number of overlay colors because of the way the precedence of the framebuffers works.

If the overlay planes contain 0 at any location, the system displays the contents of the normal framebuffer at that location. If the underlay planes contain 0 at a given location, the system displays the color at index 0 of the underlay framebuffer's color map when the normal bitplanes do not obscure them.

This loads the color map for the underlay buffers with black, red, green, blue.

```
underlay(2); /* two bitplanes, four colors */
gconfig();
drawmode(UNDERDRAW);
mapcolor(0, 0.0, 0.0, 0.0); /* black as color 0 */
mapcolor(1, 1.0, 0.0, 0.0); /* red as color 1 */
mapcolor(2, 0.0, 1.0, 0.0); /* green as color 2 */
mapcolor(3, 0.0, 0.0, 1.0); /* blue as color 3 */
```

10.5 Cursor Techniques

The cursor is handled with special cursor hardware. When the color guns scan the screen, they look at the cursor mask to determine what color to draw the cursor with as they cross the square region of the screen where the cursor is to be drawn. The cursor mask can be 1 or 2 bits deep. If the cursor mask is zero, the normal color is presented. If the mask is nonzero, the mask value is looked up in a color table (similar to overlay) to find out which color to draw. The cursor color takes precedence over even the overlay color. As with overlays, if the cursor mask is 1 bit deep, there is only one possible color; if it is 2 bits deep, the cursor can have up to three colors.

10.5.1 Types of Cursors

The system supports five cursor types: a 16×16-bit cursor in one or three colors, a 32×32-bit cursor in one or three colors, and a cross-hair one-color cursor. To specify a cursor completely, you need to specify not only its type, but its shape and color(s). In addition, every cursor has an origin, or "hot spot," and can be turned on or off.

Default Cursor

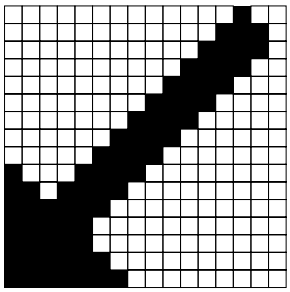
There is a default cursor, cursor number zero (0), which is an arrow pointing to the upper-left corner of the cursor glyph, and whose origin is at (0, 15), the tip of the arrow. The default cursor (number 0) cannot be redefined, and can always be used. The position of the origin of the cursor, or the cursor's hot spot, is set to the current values of the valuators that are attached to the cursor.

Cross-Hair Cursor

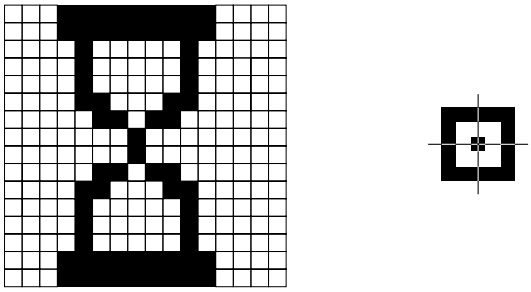
The cross-hair cursor (`CCROSS`) is formed with 1-pixel wide intersecting horizontal and vertical lines that extend completely across the screen. It is a one-color cursor that always uses cursor color 3 as its color. Its origin is at the intersection of the two lines; the default center is (15, 15). The hot spot is at the center of the cross.

The cross-hair cursor is formed from a default glyph that cannot be changed. If you assign a value to it with `defcursor()`, the user-defined glyph is ignored. The color of the cross-hair cursor is set by mapping color index 3.

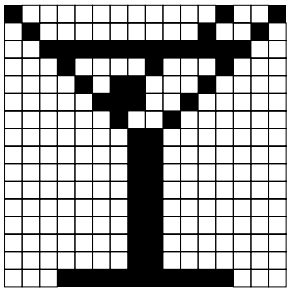
Figure 10-2 shows some example cursors.



```
Cursor arrow = {0xFE00,  0xFC00,  0xF800,  0xF800,
                0xFC00,  0xDE00,  0x8F00,  0x0780,
                0x03C0,  0x01E0,  0x00F0,  0x0078,
                0x003C,  0x001E,  0x000E,  0x0004}
```



```
Cursor hourglass = {0x1FF0,  0x1FF0,  0x0820,  0x0820,
                    0x0820,  0x0C60,  0x06C0,  0x0100,
                    0x0100,  0x06C0,  0x0C60,  0x0820,
                    0x0820,  0x0820,  0x1FF0,  0x1FF0}
```



```
Cursor martini = {0x1FF8,  0x0180,  0x0180,  0x0180,
                  0x0180,  0x0180,  0x0180,  0x0180,
                  0x0180,  0x0240,  0x0720,  0x0B10,
                  0x1088,  0x3FFc,  0x4022,  0x8011}
```

Figure 10-2 Example Cursors

10.5.2 Creating and Using Cursors

To define and use a new cursor, follow these steps:

1. Set the cursor type to one of the five allowable types with `curstype()`.
2. Define the cursor's shape and assign it a number with `defcursor()`.
3. If necessary, define its origin (or hot spot) with `curorigin()`, and its color(s) with `drawmode()` and `mapcolor()`.
4. Finally, the new cursor becomes the current cursor with a call to `setcursor()`.

If an application needs a number of different cursors, it typically defines all of them on initialization, then switches from one to another using `setcursor()` (and perhaps `mapcolor()`). Although they do not physically do so, cursors can be thought of as occupying 1 or 2 bitplanes of their own, which behave like overlay bitplanes as described above. A one-color cursor uses one bitplane, and a three-color cursor occupies two. Where there are 0s in the cursor's bitplane(s), the contents of the standard, overlay, and underlay bitplanes appear. In the same way that overlay colors are defined, `drawmode()` and `mapcolor()` define the cursor's color(s).

For a one-color cursor, first, call:

```
drawmode(CURSORDRAW)
```

followed by:

```
mapcolor(1, r, g, b)
```

For a three-color cursor, call:

```
mapcolor(1, r1, g1, b1)
mapcolor(2, r2, g2, b2)
mapcolor(3, r3, g3, b3)
```

Note: Three-color cursors might not be supported on all future versions of hardware. To write code that is portable, use only single-color cursors.

Whenever the cursor pattern (described below) contains a 1(=01), (r1, r_1, g_1, b_1) is presented; when it is 2(=10), (r2, r_2, g_2, b_2) appears, and so on. Be sure to call `drawmode(NORMALDRAW)` after you have defined the cursor's colors.

10.5.3 Cursor Subroutines

This section describes the cursor subroutines.

curstype

`curstype()` defines the current cursor type:

```
void curstype(long typ)
```

type is one of `C16X1`, `C16X2`, `C32X1`, `C32X2`, and `CCROSS`. It is used by `defcursor()` to determine the dimensions of the arrays that define the cursor's shape. `C16X1` is the default value.

After you call `curstype()`, call `defcursor()` to specify the appropriately sized array and to assign a numeric value to the cursor glyph.

defcursor

`defcursor()` defines a cursor glyph:

```
void defcursor(short n, Cursor curs)
```

The index *n* defines the cursor number, and *curs* is an array of bits of the correct size, depending on the current cursor type. The format of the array of bits is exactly the same as that for characters in a font—the 16-bit word at the lower-left is given first, then (if the cursor is 32 bits wide) the word to its right. Continue in this way to the top of the cursor for either 16 or 64 words. If the cursor is three-colored, another set of 16 or 64 words follows, again beginning at the bottom, for the second plane of the mask.

curorigin

`curorigin()` sets the origin of a cursor:

```
void curorigin (short n, short xorigin, short yorigin)
```

The origin is the point on the cursor that aligns with the current cursor valuators. The lower-left corner of the cursor has coordinates (0,0). Before calling `curorigin()`, you must define the cursor with `defcursor()`. The number *n* is an index into the cursor table created by `defcursor()`. `curorigin()` does not take effect until there a a call to `setcursor()`.

setcursor

`setcursor()` sets the cursor characteristics:

```
void setcursor(short index, Colorindex color, Colorindex wtm)
```

It selects a cursor glyph from among those defined with `defcursor()`. *index* picks a glyph from the definition table. *color* and *wtm* are ignored. They are present for compatibility with older systems that made use of them. Set the color for the cursor with `mapcolor()` and `drawmode()`.

getcursor

`getcursor()` returns the cursor characteristics:

```
void getcursor(short *index, Colorindex *color, Colorindex *wtm, Boolean b)
```

It returns two values: the cursor glyph (*index*) and a boolean value (*b*), which indicates whether the cursor is visible.

The default is the glyph index 0 in the cursor table, displayed with the color 1, drawn in the first available bitplane, and automatically updated on each vertical retrace.

10.5.4 Sample Cursor Program

This sample program, *flag.c*, defines a three-color 32×32 cursor in the shape of a United States flag. Unfortunately, 32×32 is small, so there is room for only 12 stars. (Note that a three-color cursor is not supported on the Personal IRIS; hence, `C16X2` and `C32X2` cursor types are not available on the Personal IRIS.)

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

unsigned short curs2[128] = {
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
```

```
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0xffff, 0x6666, 0xffff,
    0x6666, 0xffff, 0x0000, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0x0000, 0x0000, 0x0000, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0x0000, 0xffff, 0x0000
};

main()
{
    short val;
    prefsize(400, 400);
    if (getgdesc(GD_BITS_CURSOR) < 2) {
        fprintf(stderr, "2-plane cursor not available\n");
        return 1;
    }
    winopen("flag");
    color(BLACK);
    clear();
    qdevice(ESCKEY);
    drawmode(CURSORDRAW);
    mapcolor(1, 255, 0, 0);
    mapcolor(2, 0, 0, 255);
    mapcolor(3, 255, 255, 255);
    drawmode(NORMALDRAW);
    curstype(C32X2);
    defcursor(1, curs2);
```

```
    setcursor(1, 0, 0);
    while (TRUE) {
    Device dev;
    dev = qread(&val);
        if (dev == ESCKEY && val == 0);
            break;
        if (dev == INPUTCHANGE &&val != 0) {
            drawmode(CURSORDRAW);
            mapcolor(1,255,0,0);
            mapcolor(2,0,0,255);
            mapcolor(3,255,255,255);
            drawmode(NORMALDRAW);
        }
    }
    gexit();
    return 0;
}
```