sun® microsystems

4.1 Pixrect Reference Manual

Part Number: 800-4835-10
Revision A of 5 January, 1990

Contents

Tables

Figures

Preface

This document describes the Pixrect Graphics Library, a low-level raster operations library for writing device-independent applications for Sun products.

This document is not intended to be a tutorial on writing application programs with the Pixrect library. The reader should be familiar with the C programming language, and have access to some of the references listed below on raster graphics.

Audience

The intended reader of this document is an applications programmer who is familiar with interactive computer graphics and the C programming language. This manual contains several example programs that can be used as templates for larger pixrect applications.

Documentation Conventions

Italic font signifies file names, function arguments, variables and internal states of pixrect. *Italic font* is also used to emphasize important words and phrases. SMALL CAPS indicate values in enumerated types. `Listing font` is used for function names.

References to manual pages are shown with the name of the man page in listing font, followed by the manual page section in parenthesis: `ls`(1).

Two types of *dialogue boxes* are used in the manual. White boxes show example output and programs. Gray boxes show interactive sessions with the computer. To distinguish between computer and user output, computer output is shown in `listing font`, while user input is in **`bold listing font`**.

References

SunView 1 Programmer's Guide.
(Sun Publication Number: 800-1783)

SunView 1 System Programmer's Guide.
(Sun Publication Number: 800-1784)

Writing Device Drivers
(Sun Publication Number: 800-3851)

Conrac Corporation. *Raster Graphics Handbook*, Second Edition. Van Nostrand Reinhold, 1985.

J.D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

D. Ingalls. *Smalltalk Graphics Kernel.* Byte, August 1981.

B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

R. Pike, Leo Guibas, Dan Ingalls. *Bitmap Graphics.* ACM/SIGGRAPH 1984 Conference Course Notes. IEEE Computer Graphics and Applications, April 1984.

David F. Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill, 1985.

1

Introduction

1

Introduction

This document describes the *Pixrect Graphics Library*, a set of routines that manipulates rectangular areas of pixels either on screen or in memory. These routines, called raster operations, or *RasterOps*, are common to all Sun workstations. They allow application programmers to manipulate the bit-mapped display on any Sun workstation.

From a software perspective, the Pixrect Graphics Library is a low-level graphics package, sitting on top of the display device drivers. For most applications, the higher-level abstractions available in SunView and the Sun graphic standards libraries are a more appropriate interface (see the preface of this manual for references).

Limitations

The Pixrect Library is intended only for accessing and manipulating two-dimensional rectangular regions of a display device in a device-independent fashion.

Windows
: The Pixrect Library does not support overlapping windows. These can be implemented with memory pixrects by the application, but the SunView package already offers a sophisticated, easy-to-use programming interface for this purpose.

Input Devices
: The Pixrect Library does not have input functions. An application can use the input functions available in SunView, or make system calls directly to the raw input devices (see `mouse`(4) and `kbd`(4)).

Functionality
: The Pixrect library doesn't support any type of display list, lighting model, 3-d, transformations, etc.

1.1. Overview

Each chapter describes a major feature of the Pixrect Library.

- This chapter introduces the Pixrect Library, defines important terms and concepts, and describes the resources available to the programmer.
- Chapter 2 explains how to write pixrect programs that can run on all Sun systems.

- Chapter 3 covers the operations for *opening and manipulating* pixrects.
- Chapter 4 describes the *text facilities* in the Pixrect Library.
- Chapter 5 discusses *memory pixrects*, rectangular regions of virtual memory that are manipulated as pixrects.
- Chapter 6 explains the *file I/O* (Input/Output) functions in the Pixrect Library. These functions can serve to store and retrieve pixrects from disk files.
- Appendix A is a guide for writing *pixrect device drivers*.
- Appendix B lists the *functions and macros* in the Pixrect Library.
- Appendix C lists the *types and structures* in the Pixrect Library.

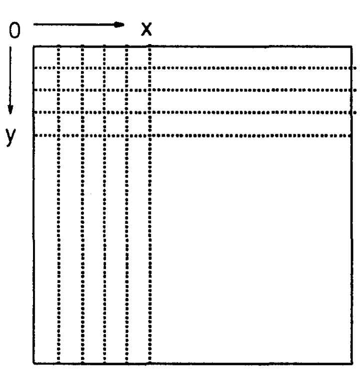
1.2. Important Concepts

This section describes some of the important concepts behind the Pixrect Library. It is not intended to be complete, but rather to explain some features of the Pixrect Library that make it unique among graphics packages.

Pixrects

A *pixrect* is the graphics analogy to an instance of a *class* used in object-oriented programming languages. It consists of bitmap data and the operations that can be performed on that data. The implementation of the operations and the data itself is hidden from the programmer (the only exception is memory pixrects, whose bitmap data can be directly manipulated. See Chapter 5 for details.) The pixrect is manipulated by using one of the functions in the Pixrect Library that is valid for that pixrect, which is analogous to sending it a message in object-oriented programming.

A pixrect object can reside on a variety of devices, including memory, different types of graphics displays and printers. Since the available operations are the same regardless of the device in which the pixrect resides, the programmer can ignore device particularities while writing the application.



Screen Coordinates

The *screen coordinate* system is two-dimensional. The origin is in the upper left corner, with x and y increasing to the right and down. The coordinates describing pixel locations in a pixrect are integers ranging from 0 to the pixrect's width (for x) or height (for y) minus 1. The maximum value for x and y is 32767.

Pixels

A *pixel* is the smallest individual picture element that can be displayed on the screen. A pixel consists of an address (corresponding to an x and y coordinate) that specifies the pixel, and a value that controls the color displayed. The pixel address can be absolute (its screen coordinate), or relative to some rectangular sub-region of the screen. A pixel has a depth (the number of bits it contains) that determines the range of colors it can display. A single-bit pixel can be only black or white, and is used in monochrome displays. Pixels with more bits can display grayscale values or color. The most common pixel depths are 1, 8, 16, or 24 bits per pixel.

Bitmaps

A *bitmap* is a rectangular region of screen space. Each pixel on the screen corresponds to some number of bits in the screen memory. The value of these bits determines the color of the corresponding pixel. These groups are arranged in an array that can be accessed using the x and y coordinates of the corresponding pixel. A pixrect bitmap can be up to 32767 pixels wide, and up to 32767 pixels high.

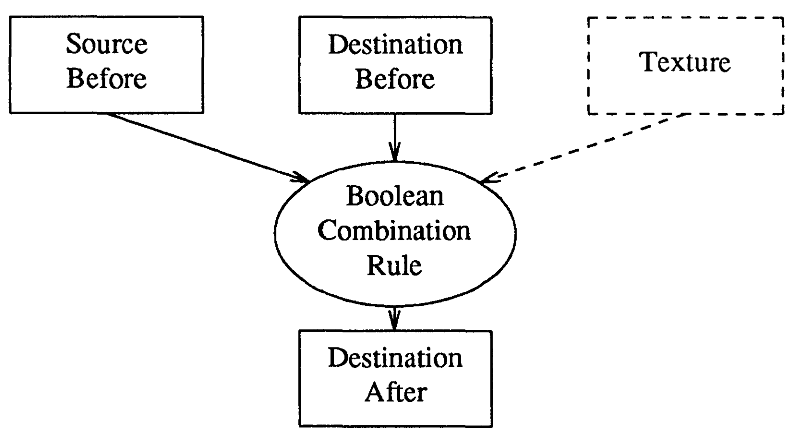
The word "bitmap" can describe the type of display, indicating that it uses raster instead of vector display technology, for instance. More commonly, it refers to the images stored in bitmap format. Examples of the second type of bitmap include the screen image, window images, the cursor, and icons.

RasterOps

RasterOps are the legal operations available for modifying pixrects. A RasterOp is an operation that takes two bitmaps as arguments: a *source* bitmap and the current state of the *destination* bitmap. The RasterOp then performs a boolean operation using these arguments, pixel by pixel, and writes the final result to the destination bitmap. The source bitmap may be a pattern, or it may be defined as a region of some constant value.

The `pr_stencil()` function is the only RasterOp that breaks this rule. Along with the source and destination bitmaps, this function takes an additional argument, a *texture* bitmap, and combines the three in a boolean operation. (See Chapter 3 for a more detailed explanation of the RasterOp functions available in the Pixrect Graphics Library).

Figure 1-1 *RasterOp Function*



1.3. Using Pixrects

The procedure for drawing pictures using pixrects requires three basic steps:

1. Opening a pixrect object.
2. Drawing a picture into the pixrect, using the set of valid operations for that particular pixrect type. Example operations could include:

```
pr_put()
pr_vector()
pr_rop()
.
.
.
```

3. Closing the pixrect.

Primary Pixrect

If the pixrect resides on a display device, the result of each drawing operation becomes visible immediately. Opening a display pixrect does not erase the previous contents of the display. Closing the pixrect also has no effect on the contents of the display.

Secondary Pixrect

A secondary pixrect is a proper subset of its parent pixrect. The results of drawing operations to a secondary pixrect are displayed if the parent's pixrect is visible, and the output is within the bounds of the secondary pixrect's clipping window. A secondary pixrect can simplify programming by allowing the programmer to isolate a section of a larger pixrect, thus sending drawing commands relative to that pixrect's coordinate system, rather than to its parent's. Pixrects can be nested to any depth.

Memory Pixrect

A memory pixrect allocates a section of memory in the workstation. Unlike a primary or secondary pixrect, a memory pixrect clears its bitmap to zeros when opened. Operations performed on memory pixrects do not show on the screen. An image in a memory pixrect can be copied to a display pixrect, which is a simple form of double buffering. A memory pixrect can also serve as a buffer or scratch pad, storing bitmaps for later use or saving the results of previous operations.

Basic Example

The following example draws a diagonal line near the upper left corner of the workstation's default display.

Figure 1-2 *Basic Example Program*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *screen;

    screen = pr_open("/dev/fb");
    pr_vector(screen, 10, 20, 70, 80, PIX_SET, 1);
    pr_close(screen);
}
```

The header file will `<pixrect/pixrect_hs.h>` `#include` all of the header files necessary for working with the functions, macros, and data structures in the Pixrect Library.

Compiling

The example program can be compiled as follows:

```
example% cc line.c -o line -lpixrect
```

This command line compiles the program in `line.c`. The `-lpixrect` option causes the C compiler to link the Pixrect Library to the application program and to create an executable file named `line`.

The example program can be executed by the SunOS C-shell:

```
example% line
```

A diagonal line appears in the upper left-hand corner of the screen.

Pixrect `lint` Library

The Pixrect Library provides a `lint`(1) library, which allows `lint` to check your program beyond the capabilities of the C compiler. Using the `-lpixrect` flag provides `lint` with pixrect-specific information that prevents bogus error messages. You could use `lint` to check the example program with a command like this:

```
example% lint line.c -lpixrect
```

Note that most of the error messages generated by `lint` are warnings, and may not necessarily have any effect on the operation of the program. For a detailed explanation of `lint`, see the discussion on `lint` in the *C Programmer's Guide*.

1.4. Pixrect Data Structures

All of the important pixrect data structures are stored in the header files shown in the table below. They can be found in the /usr/include/pixrect directory. Use these files to find the definition of a function or macro.

Table 1-1 *Pixrect Header Files*

pixrect_hs.h	#includes all pixrect files
pixrect.h	most pixrect definitions
memvar.h	memory pixrects
pixfont.h	text operations
traprop.h	traprop definitions
pr_line.h	defines wide and textured vectors
pr_planegroups.h	frame buffers
pr_util.h	internal definitions

2

Portability Considerations

2

Portability Considerations

This chapter addresses pixrect portability among the various Sun architectures. Since Pixrect is a low-level graphics library, it is not completely device independent. Currently, the Sun386i is the only Sun architecture for which porting is an issue. (It is the first Sun system to use the Intel 80386 processor.) The pixrect software has been designed to minimize porting difficulties; nevertheless, there are some portability factors to take into consideration.

The sections below describe the portability problems caused by the Sun386i system, and their solutions.

2.1. Byte Ordering

The 80386, 680*X*0 (where *X* is either 2 or 3), and SPARC are 32-bit processors. This means that all data read or written by these processors pass through 32-bit wide registers. The order in which the data — the bytes and bits — are arranged in the 80386's registers differs from the 680*X*0 and SPARC families. These differences are illustrated in the figure below:

Figure 2-1 *Byte and Bit Ordering in the 80386, 680X0 and SPARC*

80386

<table>
<tr><td>31 30 29 28 27 26 25 24</td><td>23 22 21 20 19 18 17 16</td><td>15 14 13 12 11 10 09 08</td><td>07 06 05 04 03 02 01 00</td></tr>
<tr><td>Byte n+3</td><td>Byte n+2</td><td>Byte n+1</td><td>Byte n</td></tr>
<tr><td colspan="2">Word n+1</td><td colspan="2">Word n</td></tr>
<tr><td colspan="4">Doubleword n</td></tr>
</table>

680*X*0 and SPARC

<table>
<tr><td>31 30 29 28 27 26 25 24</td><td>23 22 21 20 19 18 17 16</td><td>15 14 13 12 11 10 09 08</td><td>07 06 05 04 03 02 01 00</td></tr>
<tr><td>Byte n</td><td>Byte n+1</td><td>Byte n+2</td><td>Byte n+3</td></tr>
<tr><td colspan="2">Word n</td><td colspan="2">Word n+1</td></tr>
<tr><td colspan="4">Longword n</td></tr>
</table>

Byte Swapping and Bit Flipping

The Sun386i is based on the 80386 processor, which handles byte ordering differently than 680*X*0 and SPARC processors. This affects the Sun386i's interpretation of graphics files — font files, icon files, cursor files, and screendumps — generated by the other two architectures. Typically, frame buffers are accessed as if they were word (16-bit integer) width devices, or as the device appearing to be an array of words.

Because the byte ordering of words differs on the two architectures, transferring a graphics file from one to the other usually results in a garbled picture.

On the 680*X*0 monochrome frame buffer, the bits are shifted out of the word starting at the most significant bit — bit 15. The upper left-most pixel on the screen is word 0, bit 15 of the frame buffer memory. The next pixel, scanning from left to right as you view the screen, is bit 14. The pixel to the right of the first 16 pixels displayed comes from word 1, bit 15. When interpreted as integers, the most significant and least significant bytes are:

680*X*0

	MSB	LSB
word 0	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
word 1	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
. . .	. . .	. . .
word n	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

For example, the integer (word) value `0x370D` in word 0 appears on the 680*X*0 and SPARC monochrome frame buffer as the pixel sequence:
□□■■□■■■□□□□■■□■.

On the 80386 monochrome frame buffer, the bits are shifted out of the word from the least significant bit — bit 0 — to the most — bit 15:

80386

	LSB	MSB
word 0	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
word 1	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
. . .	. . .	. . .
word n	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15

For example, the integer (word) value `0x370D` in word 0 appears on the screen with the 80386 frame buffer as the pixel sequence: ■□■■□□□□■■■□■■□□.

The bytes are backward and the bits are in the opposite order. Since a graphics file is usually generated as an array of words, the bytes are backward for a typical 80386 frame buffer when handling files generated by 680*X*0 and SPARC machines. Eight-bit color frame buffers represent each pixel as a byte of data, so the bit order is already correct; conversion requires only byte swapping.

For monochrome frame buffers, each pixel is represented by a single bit. Scanning from right to left presents a bit flip and byte swap problem. The right-most (low-order) bit of a bit field now represents the left-most pixel on the screen.

Because of the large number of existing files that use it, the 680*X*0/SPARC format is the standard format for describing graphics images on all Sun systems. This eliminates the need for two sets of files in a mixed-architecture network. Consequently, if you are porting programs to the Sun386i from other Sun systems — programs that access the frame buffer through SunView and Pixrect — byte and bit ordering are handled automatically at run time. The 680*X*0/SPARC format images are converted to 80386 format.

2.2. Flipping Pixrects

Sun386i systems convert 680X0/SPARC format images into 80386 format just before they are used. The procedure that converts them is a new pixrect routine, `pr_flip()`, found only in the Sun386i version of Pixrect.

The internal data of a pixrect is referred to by its `pr_data` field.

```
typedef struct pixrect {
        struct  pixrectops *pr_ops;
        struct  pr_size pr_size;
        int     pr_depth;
        caddr_t pr_data;        /*pointer to mpr*/
} Pixrect;
```

If it is a memory pixrect, the structure referred to by `pr_data` is:

```
struct mpr_data {
        int     md_linebytes;
        short   *md_image;
        struct  pr_pos md_offset;
        short   md_primary;
        short   md_flags;       /*flag bits*/
     };
```

There are two new flag bits in the `md_flags` word that control the operation of `pr_flip()`. The flags `MP_REVERSEVIDEO`, `MP_DISPLAY`, and `MP_PLANEMASK` are now followed by `MP_I386` and `MP_STATIC`. If *true*, `MP_I386` indicates that the pixrect in question is already in Sun386i (80386) display format; that is, it has already been modified by `pr_flip()`. If `MP_STATIC` is *true*, the pixrect in question is a static pixrect. (In practice, this flag is sometimes set for other purposes as well.)

The `pr_flip()` Routine

The `pr_flip()` routine operates on individual pixrects. It takes one argument, a pointer to a pixrect structure, and returns void. When called, it first checks to see if the pixrect has already been flipped (`MP_I386` == TRUE). If not, it flips the image area, 16 bits at a time. First the bit order is reversed, then the bytes are swapped. The `pr_flip()` does not flip a display pixrect or a secondary pixrect unless the pixrect is static, that is, `MP_STATIC` == TRUE.

When a pixrect is modified by a `pr_flip()` call, the changes are limited to the pixrect's image area and the state of the two new `md_flags`. The size of the pixrect structures remains unaltered. The new `md_flags` are ignored by programs running under 680X0 or SPARC.

Pixrects are flipped as they are manipulated by any of the pixrect routines listed below. As an application runs, the rate of pixrect flipping usually declines since most applications develop a *working set* of active pixrects. Pixrects that are not used are not flipped.

The routines listed below contain checkpoints where pixrects used in their arguments are examined and flipped (if necessary) by `pr_flip()`:

Table 2-1 *Routines that call* `pr_flip()`

`mem_rop()`
`mem_create()`
`pr_region()`
`pr_vector()`
`pr_dump_init()`
`pf_open()`
`pf_open_private()`
`pr_stencil()`
`pr_batchrop()`
`pr_replrop()`
`pr_get()`
`pr_put()`
`pr_load()`
`pr_dump()`
`icon_display()`
`DEFINE_ICON_FROM_IMAGE`

NOTE *Icons are either static or created with* `icon_load()`. *Static icons can be created with* DEFINE_ICON_FROM_IMAGE. *Both of these SunView features are described in the SunView 1 Programmer's Guide.*

Fonts are converted by the `pf_open()` *or* `pf_open_private()` *routines. No other conversions are allowed. The libraries work only with the existing standard font files.*

Guidelines for Sun386i Systems

1. Check code that draws manually into a pixrect. It may not work properly on a Sun386i without modification. The modification required depends on the particulars of the drawing operation.
2. Manual operations (not involving `libpixrect` routines) should be performed on a pixrect **before** converting it to 80386 format.
3. `mem_create()` creates an 80386-format pixrect on Sun386i machines.
4. `mem_point` does **not** set the `MP_I386` flag. The pixrect is still marked **not** flipped.

5. To create an icon, use `mem_point()` to make a pixrect connected to an existing static image or to an image that you have created dynamically.

6. Use `DEFINE_ICON_FROM_IMAGE` (SunView) to create static icons. All static icons are initially created in 680*X*0/SPARC format. They are converted to 80386 format when they are involved in a raster operation.

3

Pixrect Operations

3

Pixrect Operations

Pixrect objects contain procedures to perform the following operations:

- create or destroy a pixrect— `pr_open()`, `pr_region()`, and `pr_destroy()`.
- read and write the values of single pixels within a pixrect— `pr_get()` and `pr_put()`.
- use RasterOp functions to simultaneously affect multiple pixels within a pixrect:

  `pr_rop`
  : write from a source pixrect to a destination pixrect

  `pr_stencil`
  : write from a source pixrect to a destination pixrect through a mask pixrect

  `pr_replrop`
  : replicate a constant source pixrect pattern throughout a destination pixrect

  `pr_batchrop`
  : write a batch of source pixrects to a sequence of locations within a single destination pixrect

  `pr_vector`, `pr_line`
  : draw a straight line in a pixrect

  `pr_polygon_2`
  : draw a polygon in a pixrect

- draw text (described in Chapter 4, *Text Facilities for Pixrects*).
- read/write the display's colormap (`pr_getcolormap()`, `pr_putcolormap()`)
- select particular bit-planes in a color pixrect's bitmap for manipulation— `pr_getattributes()`, `pr_putattributes()`
- control hardware double buffering— `pr_dbl_get()` and `pr_dbl_set()`.

From an object-oriented viewpoint, all pixrects contain both data and procedures to manipulate its data, which allow them to be device-independent. The pixrect uses the function appropriate to its environment when asked to perform an operation.

From the programmer's point of view, pixrects are manipulated using procedure calls embedded in the application program. Internally, the pixrect procedures that behave the same for all pixrects are implemented by a single procedure, to make them more efficient. The device-dependent calls are macros that access the appropriate procedure within the pixrect object. This is almost equivalent to passing the pixrect object a message, which causes the pixrect to invoke the appropriate method (procedure).

Each pixrect object includes an internal pointer to a `pixrectops` structure that holds the addresses of the particular device-dependent procedures appropriate to that pixrect. Clients may access these procedures in a device-independent fashion by calling the procedure through the `pixrectops` structure, rather than executing the procedure directly. To simplify this indirection, the Pixrect Library provides a set of macros that resemble simple procedure calls to generic operations. These macros expand to invocations of the corresponding procedure in the `pixrectops` structure.

In this manual, the description of each operation specifies whether it is a true procedure or a macro, since some of the arguments to macros are expanded multiple times and could cause errors if the arguments contain expressions with side effects. (In fact, there are two sets of parallel macros, which differ only in how their arguments use the geometry data structures.)

3.1. The `pixrectops` Structure

```
struct pixrectops {
    int (*pro_rop)();
    int (*pro_stencil)();
    int (*pro_batchrop)();
    int (*pro_nop)();
    int (*pro_destroy)();
    int (*pro_get)();
    int (*pro_put)();
    int (*pro_vector)();
    Pixrect *(*pro_region)();
    int (*pro_putcolormap)();
    int (*pro_getcolormap)();
    int (*pro_putattributes)();
    int (*pro_getattributes)();
};
```

The `pixrectops` structure is a collection of pointers to the device-dependent procedures for a particular device. All other operations are implemented by device-independent procedures. From an object-oriented point of view, this structure provides the procedural interface to the pixrect object, translating messages to methods. This structure is designed to allow expansion; additional functions may be added in future releases.

3.2. Calling Pixrect Procedures

A pixrect procedure normally expects a number of arguments. These arguments can include: a pointer to the pixrect being manipulated, the dimensions and offset of a subregion within a pixrect, and an `op` argument describing the operation to be performed. This section describes these arguments in detail, and the results returned by the pixrect procedure.

Argument Conventions

In this manual, the conventions listed in Table 3-1 are used in naming the arguments to pixrect operations.

Table 3-1 *Argument Name Conventions*

Argument	*Meaning*
d*suffix*	destination
s*suffix*	source
*prefix***x**	offset to left edge of pixrect
*prefix***y**	offset to top edge of pixrect
*prefix***w**	width of pixrect (0 to 32767)
*prefix***h**	height of pixrect (0 to 32767)

The `x` and `y` values given to functions that operate on a pixrect must be within the boundaries of that pixrect, and must be in the range 0 to 32767.

Pixrect Errors

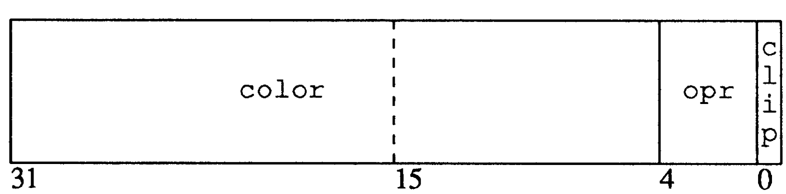
Pixrect operations indicate an error condition in one of two ways, depending on the type of value the operation normally returns. Pixrect operations that return a pointer to a structure return NULL when they fail. For pixrect operations that return an integer status code, a return value of `PIX_ERR == -1` indicates failure, while 0 indicates that the procedure completed successfully. The description of each pixrect procedure makes note of any exceptions to this convention.

3.3. The Op Argument

The multi-pixel operations described in the next section all use a uniform mechanism for specifying the operation that is to produce destination pixel values. This operation, given in the `op` argument, includes several components:

- A single constant source value may be specified in the color field, bits 5 – 31 of the `op` argument.
- A RasterOp function is specified in the operation field, bits 1 – 4 of the `op` argument.
- Clipping, which is normally performed by every pixrect operation, may be turned off by setting the `PIX_DONTCLIP` flag (bit 0) in the `op` argument.

Figure 3-1 *Structure of an* `op` *Argument*



Specifying a RasterOp Function

Four bits of the operation field (opr in figure 3-1) in the op argument are used to specify one of the 16 distinct logical functions that combine monochrome source and destination pixels to give a monochrome result. This encoding is generalized to pixels of arbitrary depth by specifying that the function is applied to corresponding bits of the pixels in parallel. Some functions are much more common than others; the most useful are identified in Table 3-2.

A convenient and intelligible form of encoding the function into four bits is supported by the following definitions:

```
#define PIX_SRC (0xC << 1)
#define PIX_DST (0xA << 1)
#define PIX_NOT(op)  ((op) ^ 0x1E)
```

PIX_SRC and PIX_DST are defined constants, while PIX_NOT is a macro. Together, they allow the desired function to be specified by performing the corresponding logical operations on the appropriate constants.

NOTE *If you want to use the ones complement (~) operator in your program to perform negation in a raster operation, it must be used in conjunction with the* PIX_NOT *macro.*

A particular application of these logical operations allows definition of PIX_SET and PIX_CLR operations. The definition of the PIX_SET operation that follows is always true, and hence sets this result:

```
#define PIX_SET (0xF << 1)
```

The definition of the PIX_CLR operation is always false, and hence clears this result:

```
#define PIX_CLR (0 << 1)
```

Other common RasterOp functions are defined in the following table:

Table 3-2 *Useful Combinations of RasterOps*

Op with Value	***Result***	
PIX_SRC	*write*	same as source argument
PIX_DST	*no-op*	same as destination argument
PIX_SRC | PIX_DST	*paint*	OR of source and destination
PIX_SRC & PIX_DST	*mask*	AND of source and destination
PIX_NOT(PIX_SRC) & PIX_DST	*erase*	AND destination with source negation
PIX_NOT(PIX_DST)	*invert area*	negate the existing values
PIX_SRC ^ PIX_DST	*inverting paint*	XOR of source and destination

Specifying a Color

A single color value can be encoded in bits 5-31 of the `op` argument. The following macro supports this encoding:

```
#define PIX_COLOR(color)     ((color) << 5)
```

Another macro extracts the color field from an encoded op:

```
#define PIX_OPCOLOR(op)      ((op) >> 5)
```

NOTE *The color is not part of the function component of the* `op` *argument and should never be part of an argument to* `PIX_NOT`.

The specified color is used by pixrect functions in two situations:

1. If the source pixrect argument is NULL, the source is a constant pixel value, and the RasterOp source operand is treated as an infinite rectangle of pixels with the specified color.
2. If the source pixrect has a depth of 1 bit and the destination pixrect has a greater depth, the RasterOp source operand is the specified color for each "1" source pixel and zero for each "0" source pixel. A color of zero is treated as a special case; it is converted to the maximum pixel value for the destination pixrect.

If the destination pixrect has a depth of 1 bit, any nonzero color value is treated as 1; for other depths, less significant bits of the color value are used. If the destination pixrect is 32-bits deep, the encoded color is sign extended.

Op Arguments between Pixrects of Different Depths

The standard *rop* operations are allowed, to a limited extent, between pixrects of different depths. The following table sums up the limitations.

Table 3-3 *rop Operations (depth limitations)*

Source	Destination: 0	1	8	24	32
0	—	Yes	Yes		Yes
1	—	Yes	Yes		Yes
8	—	No	Yes		No
24	—				
32	—	No	No		Yes

The value *n* can be 1, 8, or 32 bits, but not 24 bits. Note that 8-to-32 bit and 32-to-8 bit are not supported. To translate pixel colors between 8 and 32, use the formula shown below. This format uses the 8-bit pixel value (the variable `color8`) with the 8-bit colormap to generate a 24-bit color, which is saved in the integer variable `color24`). This `color24` variable has its true color stored in `XBGR` format. The value can then be saved as a 32-bit pixel in the pixrect's `PIXPG_24BIT_COLOR` plane group.

```
int color24;
unsigned char red[256],green[256],blue[256];

color24 = red[color8] + (green[color8] << 8) + (blue[color8] << 16);
```

For a discussion of plane groups see Section 3.9.

Controlling Clipping in a RasterOp

Pixrect operations normally clip to the bounds of the operand pixrects. Sometimes this can be done more efficiently by the client at a higher level. If the client can guarantee that only pixels that should be visible are written, it may instruct the pixrect operation to bypass clipping checks, thus speeding its operation. This is done by setting the following flag in the `op` argument:

```
#define PIX_DONTCLIP 0x1
```

The result of a pixrect operation is undefined and may cause a memory fault if `PIX_DONTCLIP` is set and the operation goes out of bounds.

NOTE *The* `PIX_DONTCLIP` *flag is not part of the* `function` *component of an* `op` *argument; it should never be part of an argument to* `PIX_NOT`.

Examples of Complete Op Argument Specification

A very simple `op` argument specifies that source pixels be written to a destination, clipping to both operands:

```
op = PIX_SRC;
```

But this example would have problems with some color combinations. A better one would be:

```
op = PIX_SRC | PIX_COLOR(1);
```

A more complicated example can be used to flip the color of destination pixels between two values wherever pixels in a 1-bit source pixrect are set, with clipping disabled for maximum performance:

```
op = (PIX_DST ^ PIX_SRC) | PIX_COLOR(color1 ^ color2) \
    | PIX_DONTCLIP;
```

3.4. Creation and Destruction of Pixrects

Pixrects are created by the procedures `pr_open()` and `mem_create()`, by the procedures accessed with the macro `pr_region()`, and at compile time by the macro `mpr_static()`. Pixrects are destroyed by the procedures accessed by the macros `pr_destroy()` and `pr_close()`. The macros `mem_create()` and `mpr_static()` are for memory pixrects, and are discussed in Chapter 5. The others are described in this section.

Create a Primary Display Pixrect

```
Pixrect *pr_open(devicename)
char *devicename;
```

The properties of a non-memory pixrect depend on an underlying UNIX device. Thus, when creating the first pixrect for a device, you need to open it with a call to `pr_open()`. The default device name for your display is `/dev/fb`. (`fb` stands for *frame buffer*.) Any other device name may be used providing that it is a display device, that the kernel is configured for it, that it exists in the `/dev` directory, and that it has pixrect support. For example, devices such as the `/dev/cgsix0` device may exist on a Sun workstation, and can be opened with pixrects.

Note that `pr_open()` does not create pixrects whose pixels are stored in memory. This function is served by the procedure `mem_create()`, discussed in Chapter 5.

`pr_open()` returns a pointer to a primary pixrect structure that covers the entire surface of the named device. If it cannot, it returns NULL, and prints a message on the standard error output.

Getting Screen Parameters

To write portable programs, it is important to read the screen characteristics directly, rather than assuming them. The pixrect returned by `pr_open()` contains this information. The two most important values are the dimensions of the screen, and the depth (number of bits) of each pixel. The code sample below opens a screen pixrect, then extracts the width, height and depth (in bits) of the screen.

```
#include <pixrect/pixrect_hs.h>   include the proper definitions
#include <stdio.h>

main()
{
    Pixrect *screen, *pr_open(); screen points to screen pixrect
    int height, width, depth;    variables to make things clearer

    screen = pr_open("/dev/fb"); open the pixrect

    width  = screen->pr_size.x;  extract the data in pr_size;
    height = screen->pr_size.y;  width and height are in pixels
    depth  = screen->pr_depth;   get depth in bits

    (void)printf("width = %d, height = %d, bits/pixel = %d0,
            width, height,depth);  display result

    (void)pr_close(screen);      close the pixrect
}
```

Create Secondary Pixrect

```
#define Pixrect *pr_region(pr, x, y, w, h)
Pixrect *pr;
int x, y, w, h;

#define Pixrect *prs_region(subreg)
struct pr_subregion subreg;
```

Given an existing pixrect, it is possible to create another pixrect that refers to some or all of the pixels in the parent pixrect. This *secondary pixrect* is created by a call to the procedures invoked by the macros `pr_region()` and `prs_region()`.

The existing pixrect is addressed by `pr`; it may be a pixrect created by `pr_open()`, `mem_create()` or `mpr_static()` (a primary pixrect either on the screen, or in memory); or it may be another secondary pixrect created by a previous call to a region operation. The rectangle to be included in the new pixrect is described by `x`, `y`, `w`, and `h` in the existing pixrect. The `(x, y)` coordinates of the existing pixrect maps to the (0, 0) location in the new pixrect. If any part of the created pixrect is outside its parent, the outside part will be clipped. The `prs_region()` function does the same thing as `pr_region()`, but all of its argument values collected into the single structure `subreg`. Either region procedure will return a pointer to the new pixrect. If they fail, they return NULL.

If an existing secondary pixrect is provided in the call to the region operation, the result is another secondary pixrect referring to the underlying primary pixrect. There is no further connection between the two secondary pixrects. Generally, the distinction between primary and secondary pixrects is not important. However, no secondary pixrect should ever be used after its primary pixrect is destroyed.

Release Pixrect Resources

```
#define pr_close(pr)
Pixrect *pr;

#define pr_destroy(pr)
Pixrect *pr;

#define prs_destroy(pr)
Pixrect *pr;
```

The macros `pr_close()`, `pr_destroy()` and `prs_destroy()` invoke device-dependent procedures to destroy a pixrect, freeing resources that belong to it. The procedure returns 0 if successful, `PIX_ERR` if it fails. It may be applied to either primary or secondary pixrects. If a primary pixrect is destroyed before secondary pixrects that refer to its pixels, these secondary pixrects are invalidated; and attempting any operation other than `pr_destroy()` on them is an error. The three macros are identical; they are all defined for reasons of history and stylistic consistency.

3.5. Single-Pixel Operations

The operations `pr_get()`, `prs_get()`, `pr_put()` and `prs_put()` manipulate the value of a single pixel.

Get Pixel Value

```
#define pr_get(pr, x, y)
Pixrect *pr;
int x, y;

#define prs_get(srcprpos)
struct pr_prpos srcprpos;
```

The macros `pr_get` and `prs_get` invoke device-dependent procedures to retrieve the value of a single pixel. The `pr` argument indicates the pixrect in which the pixel can be found; `x` and `y` are the coordinates of the pixel. For `prs_get`, the same arguments are provided in the single struct `srcprpos`. The value of the pixel is returned as a 32-bit integer. If the procedure fails, it returns `PIX_ERR`.

Set Pixel Value

```
#define pr_put(pr, x, y, value)
Pixrect *pr;
int x, y, value;

#define prs_put(dstprpos, value)
struct pr_prpos dstprpos;
int value;
```

The macros `pr_put()` and `prs_put()` invoke device-dependent procedures to store a value in a single pixel. `pr` indicates the pixrect in which the pixel is to be found; `x` and `y` are the coordinates of the pixel. For `prs_put()`, the same arguments are provided in the single struct `dstprpos`. `value` is truncated on the left, if necessary, and stored in the indicated pixel. If the procedure fails, it returns `PIX_ERR`.

3.6. Multi-Pixel Operations

The following operations effect multiple pixels at one time:

- `pr_rop()`,
- `pr_stencil()`,
- `pr_replrop()`,
- `pr_batchrop()`,
- `pr_polygon_2()`, and
- `pr_vector()`.

With the exceptions of `pr_vector()` and `pr_polygon_2()`, they refer to rectangular areas of pixels. They all use a common mechanism, the `op` argument described in section *The Op Argument* to specify how pixels are to be set in the destination.

RasterOp Source to Destination

```
#define pr_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
Pixrect *dpr, *spr;
int dx, dy, dw, dh, op, sx, sy;

#define prs_rop(dstregion, op, srcprpos)
struct pr_subregion dstregion;
int op;
struct pr_prpos srcprpos;
```

The `pr_rop()` and `prs_rop()` macros invoke device-dependent procedures that perform the indicated raster operation from a source to a destination pixrect. `dpr` addresses the destination pixrect, whose pixels are affected; (`dx, dy`) is the origin (the upper-left pixel) of the affected rectangle; `dw` and `dh` are the width and height of that rectangle. `spr` specifies the source pixrect, and (`sx, sy`) specify the source origin within it. `spr` may be NULL, to indicate a constant source specified in the `op` argument, as described previously; in this case `sx` and `sy` are ignored. The `op` argument specifies the operation that is performed; its construction is described in Section 3.3.5.

`pr_rop()` is the only pixrect function that can have its source and destination as overlapping areas of the same pixrect. Doing this with any other operation generates an error.

For `prs_rop()`, the `dpr`, `dx`, `dy`, `dw` and `dh` arguments are all collected in a `pr_subregion` structure.

Raster operations are clipped to the source dimensions, if those dimensions are smaller than the destination size given. `pr_rop()` procedures return `PIX_ERR` if they fail, 0 if they succeed.

Source and destination pixrects generally must be the same depth. A major exception is monochrome (1-bit deep) pixrects. Monochrome pixrects may be a source pixrect to a destination pixrect of any depth. If the destination pixrect is not monochrome, the monochrome source pixels equal to 0 are interpreted as 0, while the source pixels equal to 1 are written in the color value given by the `op` argument of the function being used. If the color value in the `op` argument is 0, source pixels equal to 1 are written as the maximum value that can be stored in the destination pixel.

See the example program in Figure 5-2 for an illustration of `pr_rop()`.

RasterOps through a Mask

```
#define pr_stencil(dpr, dx, dy, dw, dh, op,
stpr, stx, sty, spr, sx, sy)
Pixrect *dpr, *stpr, *spr;
int dx, dy, dw, dh, op, stx, sty, sx, sy;

#define prs_stencil(dstregion, op, stenprpos, srcprpos)
struct pr_subregion dstregion;
int op;
struct pr_prpos stenprpos, srcprpos;
```

The `pr_stencil` and `prs_stencil` macros invoke device-dependent procedures that perform the indicated raster operation from a source to a destination pixrect only in areas specified by a third (stencil) pixrect. `pr_stencil()` is identical to `pr_rop()` except that the source pixrect is written through a stencil pixrect that functions as a spatial write-enable mask. The stencil pixrect must be a monochrome memory pixrect. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged. The rectangle from (`sx, sy`) in the source pixrect `spr` is aligned with the rectangle from (`stx, sty`) in the stencil pixrect `stpr`, and written to the rectangle at (`dx, dy`) with width `dw` and height `dh` in the destination pixrect `dpr`. The source pixrect `spr` may be NULL, in which case the color specified in `op` is painted through the stencil. Clipping restricts painting to the intersection of the destination, stencil, and source rectangles. `pr_stencil()` procedures return `PIX_ERR` if they fail, 0 if they succeed.

Replicating the Source Pixrect

```
pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
Pixrect *dpr, *spr;
int dx, dy, dw, dh, op, sx, sy;

#define prs_replrop(dsubreg, op, sprpos)
struct pr_subregion dsubreg;
struct pr_prpos sprpos;
```

Often the source for a raster operation consists of a pattern that is used repeatedly, or replicated to cover an area. If a single value is to be written to all pixels in the destination, the best way is to specify that value in the `color` component of a `pr_rop()` operation. But when the pattern is larger than a single pixel, a mechanism is needed for specifying the basic pattern, and how it is to be laid down repeatedly on the destination.

The `pr_replrop()` procedure replicates a source pattern repeatedly to cover a destination area. `dpr` indicates the destination pixrect. The area affected is described by the rectangle defined by `dx, dy, dw, dh`. `spr` indicates the source pixrect, and the origin within it is given by (`sx, sy`). The corresponding `prs_replrop()` macro generates a call to `pr_replrop()`, expanding its `dsubreg` into the five destination arguments, and `sprpos` into the three source arguments. `op` specifies the operation to be performed, as described above in Section 3.3, *The Op Argument.*

The effect of `pr_replrop()` is the same as if an infinite pixrect were constructed using copies of the source pixrect laid immediately adjacent to each other in both dimensions, and then a `pr_rop()` was performed from that source to the destination. For instance, a standard gray pattern may be painted across a portion of the screen by constructing a pixrect that contains exactly one tile of the pattern, and by using it as the source pixrect.

The alignment of the pattern on the destination is controlled by the source origin given by (`sx, sy`). If these values are 0, then the pattern has its origin aligned with the position in the destination given by (`dx, dy`). Another common method of alignment preserves a global alignment with the destination, for instance, in order to repair a portion of a gray pattern. In this case, the source pixel that should be aligned with the destination position is the one that has the same coordinates as that destination pixel, modulo the size of the source pixrect. `pr_replrop()` performs this modulus operation for its clients, so it suffices in this case to simply copy the destination position (dx, dy) into the source position (sx, sy).

pr_replrop() returns PIX_ERR if it fails, or 0 if it succeeds. Internally pr_replrop() may use pr_rop() procedures. In this case, pr_rop() errors are detected and returned by pr_replrop().

Multiple Source to the Same Destination

```
#define pr_batchrop(dpr, dx, dy, op, items, n)
Pixrect *dpr;
int dx, dy, op, n;
struct pr_prpos items[];

#define prs_batchrop(dstpos, op, items, n)
struct pr_prpos dstpos;
int op, n;
struct pr_prpos items[];
```

Applications such as displaying text perform the same operation from a number of source pixrects to a single destination pixrect in a fashion that is amenable to global optimization.

The pr_batchrop and prs_batchrop macros invoke device-dependent procedures that perform raster operations on a sequence of sources to successive locations in a common destination pixrect. items is an array of pr_prpos structures used by a pr_batchrop() procedure as a sequence of source pixrects. Each item in the array specifies a source pixrect and an advance in x and y. The whole of each source pixrect is used, unless it needs to be clipped to fit the destination pixrect. The advance is used to update the destination position, not as an origin in the source pixrect.

pr_batchrop() procedures take a destination specified by dpr, dx and dy, or by dstpos in the case of prs_batchrop(); an operation specified in op, as described in Section 3.3 ; and an array of pr_prpos addressed by the argument items, whose length is given in the argument n.

The destination position is initialized to the position given by dx and dy. Then, for each item, the offsets given in pos are added to the previous destination position, and the operation specified by op is performed on the source pixrect and the corresponding rectangle whose origin is at the current destination position. Note that the destination position is updated for each item in the batch, and these adjustments are cumulative.

The most common application of pr_batchrop() procedures is in painting text; additional facilities to support this application are described in Chapter 4. Note that the definition of pr_batchrop() procedures supports variable-pitch and rotated fonts, and non-Roman writing systems, as well as simple text.

pr_batchrop() procedures return PIX_ERR if they fail, 0 if they succeed. Internally, pr_batchrop() may use pr_rop() procedures. In this case, pr_rop() errors are detected and returned by pr_batchrop().

Draw Vector

```
#define pr_vector(pr, x0, y0, x1, y1, op, value)
Pixrect *pr;
int x0, y0, x1, y1, op, value;

#define prs_vector(pr, pos0, pos1, op, value)
Pixrect *pr;
struct pr_pos pos0, pos1;
int op, value;
```

The `pr_vector` and `prs_vector` macros invoke device-dependent procedures that draw a vector one unit wide between two points in the indicated pixrect. `pr_vector()` procedures draw a vector in the pixrect indicated by `pr`, with endpoints at (`x0`, `y0`) and (`x1`, `y1`), or at `pos0` and `pos1` in the case of `prs_vector()`. Portions of the vector lying outside the pixrect are clipped as long as `PIX_DONTCLIP` is 0 in the `op` argument. The `op` argument is constructed as described in Section 3.3 and `value` specifies the resulting value of pixels in the vector. There is some redundancy in this command. The value of the pixel can be specified twice; it can be set by modifying the proper bits in the `op` argument of the function, or it can be described directly with the `value` argument. In cases where both values are set, the value encoded in the `op` argument has priority. If the color in `op` is non-zero, it takes precedence over the `value` argument.

Any vector that is not vertical, horizontal or at a 45 degree angle contains *jaggies*. This phenomenon, known as *aliasing*, is due to the digital nature of the bitmap screen. It can be visualized if you imagine a vertical vector with one endpoint displaced horizontally by a single pixel. The resulting line has to jog over a pixel at some point in the traversal to the other endpoint. Balancing the vector guarantees that the jog occurs in the middle of the vector. `pr_vector()` draws *balanced* vectors. (The technique used is to balance the Bresenham error term.) The vectors are balanced according to their endpoints as given and not as clipped, so that the same pixels are drawn regardless of how the vector is clipped.

See the example program in Figure 1-2 for an illustration of `pr_vector()`.

Draw Textured Polygon

```
pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy)
Pixrect *dpr, *spr;
int dx, dy
int nbnds, npts[];
struct pr_pos *vlist;
int op, sx, sy;
```

The `pr_polygon_2()` function performs a raster operation on a polygonal area of the destination pixrect. The source can be a pattern or a constant color value.

The destination polygon is described by `nbnds`, `npts` and `vlist`. `nbnds` is the number of individual closed boundaries (vertex lists) in the polygon. A complex polygon may have one boundary for its exterior shape and several boundaries delimiting interior holes. The boundaries may intersect themselves or each other. Only those destination pixels having an odd *winding number* are painted. That is, if any line connecting a pixel to infinity crosses an odd number of boundary edges, the pixel is painted.

For each of the `nbnds` boundaries, `npts` specifies the number of points in the boundary. The `vlist` array contains the boundary points for all of the boundaries, in order. The total number of points in `vlist` is equal to the sum of the `nbnds` elements in the `npts` array. `pr_polygon_2()` automatically joins the last point and first point to close each boundary. If any boundary has fewer than 3 points, `pr_polygon_2()` returns `PIX_ERR`.

The destination coordinates `dx`, and `dy` are added to each point in `vlist`, so the same `vlist` can be used to draw polygons in different destination locations.

If the source pixrect `spr` is non-null, it is replicated in the x and y directions to cover the entire destination area. The point (`sx`, `sy`) in this extended source pixrect is aligned with the point (`dx`, `dy`) in the destination pixrect.

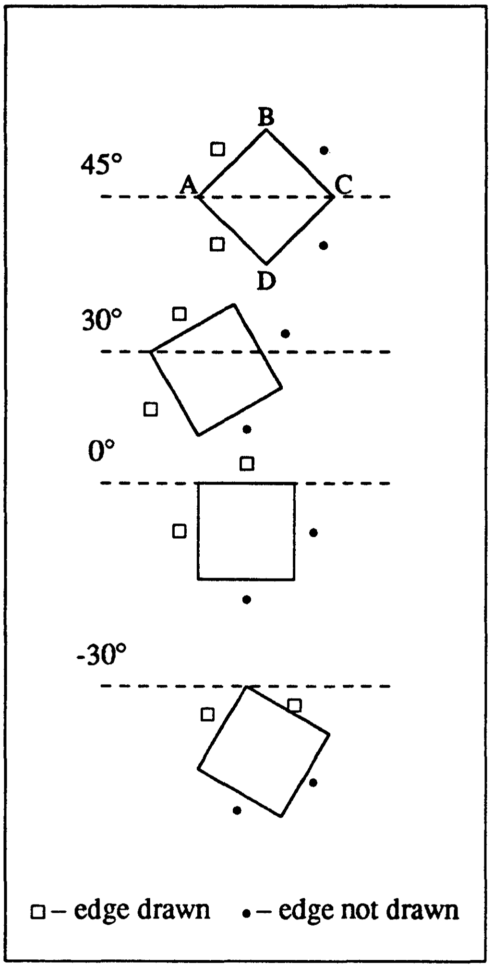
Polygons drawn by `pr_polygon_2()` are *semi-open* in the sense that on some of the edges, pixels are not drawn where a vector drawn with the same coordinates would go. Identical polygons (same size and orientation) are thus allowed to exactly tile the destination pixrect with no gaps or overlaps.

In Figure 3-3 the edges AB and DA are drawn, whereas edges BC and CD are not.

Figure 3-2 *Example Program using* `pr_polygon_2()`

```
#include <pixrect/pixrect_hs.h>

#define CENTERX(pr)  ((pr)->pr_size.x / 2)
#define NULLPR       ((Pixrect *) 0)

static struct pr_pos
    /* 45 degrees */
    vlist0[4] = { {0, 0}, { 71, -71}, {141,   0}, { 71,  71} },
    /* 30 degrees */
    vlist1[4] = { {0, 0}, { 87, -50}, {137,  37}, { 50,  87} },
    /* 0 degrees */
    vlist2[4] = { {0, 0}, {100,   0}, {100, 100}, {  0, 100} },
    /* -30 degrees */
    vlist3[4] = { {0, 0}, { 87,  50}, { 37, 137}, {-50,  87} };

main()
{
    Pixrect *pr;
    static int npts[1] = { 4 };

    if (!(pr = pr_open("/dev/fb")))
        exit(1);

    pr_polygon_2(pr, CENTERX(pr), 100, 1, npts, vlist0,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 300, 1, npts, vlist1,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 500, 1, npts, vlist2,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 700, 1, npts, vlist3,
        PIX_SET, NULLPR, 0, 0);

    pr_close(pr);
    exit(0);
}
```

Figure 3-3 *Four Polygons Drawn with* `pr_polygon_2()`



Draw Textured or Solid Lines with Width

```
#define pr_line(pr, x0, y0, x1, y1, brush, tex, op)
Pixrect *pr;
int x0, y0, x1, y1;
struct pr_brush *brush;
struct pr_texture *tex;
int op;
```

The `pr_line` macro draws a textured line based on the Bresenham line drawing algorithm, using a pen-up, pen-down approach. The programmer can define a pattern (of arbitrary length), or use a predefined default pattern (dash-dot, dotted, etc.). All pattern segments and their corresponding offsets can automatically adjust, according to the angle at which the line is drawn.

If the brush pointer is NULL, or if the width is 0 or 1, a single width vector is drawn.

The line is drawn in the pixrect indicated by `pr`, with endpoints at (`x0, y0`) and (`x1, y1`).

The `brush` field is a pointer to a structure of type `pr_brush`, which holds the width of the line segments to be rendered. The `pr_brush` structure is defined in the header file `<pixrect/pr_line.h>` as follows:

```
typedef struct pr_brush {
    int width;
} Pr_brush;
```

If the `tex` pointer is NULL, a solid vector is drawn. The `tex` field is a pointer to a structure of type `pr_texture`. The `pr_texture` structure is defined in the header file `<pixrect/pr_line.h>` as follows (fields that begin with the prefix `res_` are reserved for program internals, and are not user-definable):

```
typedef struct pr_texture {
    short *pattern;
    short offset;
    struct pr_texture_options {
        unsigned startpoint : 1,
        endpoint : 1,
        balanced : 1,
        givenpattern : 1,
        res_fat : 1,
        res_poly: 1,
        res_mvlist : 1,
        res_right : 1,
        res_close : 1;
    } options;
    short res_polyoff;
    short res_oldpatln;
    short res_fatoff;
} Pr_texture;
```

`pattern` is a pointer to an array of short integers that contain the length of each segment in the pattern. The lengths are in units of pixels. If the line is drawn at an angle, the lengths drawn are automatically adjusted (if the `givenpattern` field is set to 0) to correspond to the length of the pattern if a horizontal or vertical line was drawn. This array must be null-terminated. The first segment of the `pattern` array is assumed to be pen-down, and following segments alternate.

The addresses of the following predefined `pattern` arrays may be stored in the `pattern` field of the texture structure as well:

```
extern short pr_tex_dotted[];
extern short pr_tex_dashed[];
extern short pr_tex_dashdot[];
extern short pr_tex_dashdotdotted[];
extern short pr_tex_longdashed[];
```

The programmer-defined elements of the `pattern` array are not altered within the routine, allowing multiple calls using the same pattern. `offset` is an integer offset into the pattern, specified in pixels. Since the first segment of the `pattern` array is assumed to be pen-down, you must specify an `offset` to start on a pen-up segment. `offset` is adjusted according to the angle at which the line is drawn if the original pattern was adjusted (dependent upon the `givenpattern` bit, described later). Because of integer approximation, the adjusted `offset` can vary plus or minus one pixel from the exact adjusted `offset`.

In the options bit fields, if `startpoint` is set, the first point is always drawn, and if `endpoint` is set, the last point is drawn. If these are not specified, the line is drawn with no extra pixels set. The `balanced` bit field effectively centers the pattern within the line by computing an offset into the pattern. If the `givenpattern` bit is set, the pattern is drawn without true length correction, at any angle; this increases performance. However, the pattern of radiating lines from a common center forms concentric squares instead of circles. If the `givenpattern` bit is not set, the segment length of each element of the pattern is adjusted according to the angle at which the line is drawn. The true (angle-dependent) segment lengths are computed for one period of the pattern, using an incremental algorithm which approximates the formula:

$$angle\_pattern\_length = given\_pattern\_length * \cos(angle)$$

where all units are in pixels, and *angle* is measured from the positive x-axis. Since the algorithm angle-corrects for one period of the pattern, the longer its period, the more exact the results are.

The `op` argument specifies the raster operations used to produce destination pixel values and color.

Draw Textured or Solid Polylines with Width

```
pr_polyline(dpr, dx, dy, npts, ptlist, mvlist, brush, tex, op)
Pixrect *dpr;
int dx, dy, npts;
struct pr_pos *ptlist;
u_char *mvlist;
struct pr_brush *brush;
struct pr_texture *tex;
int op;
```

`pr_polyline` draws a polyline, or a series of disjoint polylines, using the features available in `pr_line`. The polyline is drawn in the destination pixrect indicated by `dpr`, with `dx` and `dy` being the offset into the destination pixrect for vertices to be translated in x and y, respectively. `npts` is the number of vertices in the polyline, which is always the number of lines plus 1. The `ptlist` field is an array of `npts` structures of type `pr_pos` that hold vertices. The `mvlist` field is a pointer to an array of `npts` elements, where if any element after the first is non-zero, a segment is not drawn to that vertex. The first element of the `mvlist` array controls whether the polyline(s) are automatically closed; if set, each continuous polyline is closed. If disjoint polylines are not desired (no `mvlist` is specified), the constants `POLY_CLOSE` and `POLY_DONTCLOSE` determine this behavior. `POLY_CLOSE` and `POLY_DONTCLOSE` are defined as follows:

```
#define POLY_CLOSE  ((u_char *) 1)
#define POLY_DONTCLOSE  ((u_char *) 0)
```

The brush field is a pointer to a structure of type pr_brush, and the tex field is a pointer to a structure of type pr_texture. If the tex pointer is null, a solid vector is drawn. If the brush structure is null, single-width vectors are drawn. op specifies the raster operations used to produce destination pixel values and color. brush and tex are described in detail under pr_line.

Draw Multiple Points

```
pr_polypoint(dpr, dx, dy, npts, ptlist, op)
Pixrect *dpr;
int dx, dy, npts;
struct pr_pos *ptlist;
int op;
```

The pr_polypoint routine draws an array of points on the screen under the control of the op argument. The array of points is drawn in the destination pixrect dpr, with an offset specified by the arguments dx and dy. Npts is the number of points to be rendered, and ptlist is a pointer to an array of structures of type pr_pos, that hold the vertices for each point. Color is encoded in the op argument. Portions of the array outside the pixrect are clipped unless the PIX_DONTCLIP flag is set in the op argument.

3.7. Colormap Access

A *colormap* is a table that translates a pixel value into 8-bit intensities in red, green, and blue. For a pixrect of depth n, the corresponding colormap has 2^n entries. The two most common cases are monochrome (two entries) and color (256 entries). Memory pixrects do not have colormaps.

All Sun color frame buffers display a 24-bit color value at each pixel. A 24-bit color is defined by 8-bits (256 shades) each of red, green, and blue, which produces 16.7 million different possible colors (2^{24}). Frame buffers previous to the **CG8** and **CG9** were limited in the number of different 24-bit colors that could be shown simultaneously. The **CG8** and **CG9**, however, are *true color* frame buffers. Each pixel located in the frame buffer's memory can hold an entire 24-bit color value.

Sun grayscale workstations normally use the red video signal to drive the monitor. However, when writing an application to run on a grayscale workstation, we recommend that you load the red, green, and blue components of each colormap entry with the same value to ensure that the application also runs properly on a color workstation.

Get Colormap Entries

```
#define pr_getcolormap(pr, index, count, red, green, blue)
Pixrect *pr;
int index, count;
unsigned char red[], green[], blue[];

#define prs_getcolormap(pr, index, count, red, green, blue)
Pixrect *pr;
int index, count;
unsigned char red[], green[], blue[];
```

The macros `pr_getcolormap` and `prs_getcolormap` invoke device-dependent procedures to read all or part of a colormap into arrays in memory.

These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations.

`pr` identifies the pixrect whose colormap is to be read; the `count` entries starting at `index` (zero origin) are read into the three arrays.

For monochrome pixrects the same value is read into corresponding elements of the `red`, `green` and `blue` arrays. These array elements will have their bits either all cleared, indicating `black`, or all set, indicating `white`. By default, the 0th (*background*) element is white, and the 1st (*foreground*) element is black. Colormap procedures return (-1) if the index or count are out of bounds, and 0 if they succeed.

Set Colormap Entries

```
#define pr_putcolormap(pr, index, count, red, green, blue)
Pixrect *pr;
int index, count;
unsigned char red[], green[], blue[];

#define prs_putcolormap(pr, index, count, red, green, blue)
Pixrect *pr;
int index, count;
unsigned char red[], green[], blue[];
```

The macros `pr_putcolormap` and `prs_putcolormap` invoke device-dependent procedures to store from memory into all or part of a colormap. These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations. The `count` elements starting at `index` (zero origin) in the colormap for the pixrect identified by `pr` are loaded from corresponding elements of the three arrays. For monochrome pixrects, the only value considered is `red[0]`. If this value is 0, then the pixrect is set to a dark background and light foreground. If the value is non-zero, the foreground is dark; that is, black-on-white. Monochrome pixrects are dark-on-light by default.

NOTE *Full colormap functionality is not supported for monochrome pixrects. Colormap changes to monochrome pixrects apply only to subsequent operations, whereas a colormap change to a color device instantly change all affected pixels on the display surface.*

Lookup Tables

Although 24-bit true color frame buffers have something akin to a colormap, it serves a different purpose. They are called lookup tables, and have 256 entries each of red, green, and blue. These entries affect the corresponding red, green and blue components of the displayed pixels. Lookup tables are most often used for gamma correction.

Gamma correction is the process of adjusting the color intensity values, to adjust for non-linearities in the display hardware and the human eye. Gamma corrected displays produce more realistic colors.

Because `pr_putcolormap` and `pw_putcolormap` are frequently used in existing software, they produce no errors when run on 24-bit frame buffers. The functions are simply ignored, and the lookup tables remain unchanged. To change the lookup table values, use the `pr_putlut` and `pr_getlut` commands instead.

The word *lut* is an abbreviation for *look-up table*.)

The 24-bit frame buffers also use the monochrome overlay and enable planes in a new way. Colormap commands will not work on the colormaps of these planes either. You should use the `pr_putlut` and `pr_getlut` commands to adjust their colormaps.

See the following subsection, *True Color Look-Up Table*, for definitions of `pr_putcolormap`, `pr_getcolormap`, `pr_putlut`, and `pr_getlut`.

True Color Look-Up Table

The `pr_getlut()` and `pr_putlut()` pixrect macros defined in `/usr/include/pixrect/pixrect_hs.h` read and modify the 24-bit look-up tables. They are defined as follows:

```
#include <pixrect/pixrect_hs.h>

#define pr_putlut(pr, ind, cnt, red, grn, blu)\
    (*(pr)->pr_ops->pro_putcolormap)(pr, PR_FORCE_UPDATE | ind, cnt, red, grn, blu)

#define pr_getlut(pr, ind, cnt, red, grn, blu)\
    (*(pr)->pr_ops->pro_getcolormap)(pr, PR_FORCE_UPDATE | ind, cnt, red, grn, blu)
```

Using the `pr_putlut()` macro to load the look-up tables is similar to using the `pr_putcolormap()` function. The `red[]`, `green[]`, and `blue[]` array arguments correspond to the appropriate look-up tables. Similarly, `pr_getlut()` fills these same arrays from the look-up tables.

The `PR_FORCE_UPDATE` value in the `pr_putlut()` macro is necessary because there is no colormap sharing in Pixrects. The sample program below shows how these macros are used.

```
#include <pixrect/pixrect_hs.h>

pr = pr_open("/dev/cgnine0");
pr_set_plane_group(pr, PIXPG_24BIT_COLOR); change to 24-bit plane
pr_getlut(pr, 0, 256, red, green, blue);
gamma_correct(red,green,blue); a user-supplied function...
pr_putlut(pr, 0, 256, red, green, blue);
```

This code example first opens the frame buffer and then changes the current plane group to 24-bit color (the default is the overlay plane). The `pr_putlut()` and `pr_getlut()` macros are used to read and then reload the look-up tables.

XBGR Format

The Pixrect Library already supports 1, 8, and 32-bit deep pixrects (32-bit as *true color* memory pixrects). Since true color pixrects are stored in a format that is 32-bits deep, few changes were necessary to the Pixrect Library to support the **CG8** and the **CG9**. A new pixel format called XBGR was defined to hold true color pixrects. The **CG8** and the **CG9** store 24-bit images in XBGR format:

```
#include <pixrect/pixrect_hs.h>

union fbunit {
    u_int   packed;

    struct {
        u_int   A:8;    /* high-order 8 bits unused */
        u_int   B:8;    /* bits of blue component   */
        u_int   G:8;    /* bits of green component  */
        u_int   R:8;    /* bits of red component    */
    } channel;
};
```

The 32-bit word is divided into four *channels* of 8 bits each (see the figure below). The **CG8** and **CG9** do not currently use the first channel (the high-order 8 bits). Its value is undefined, and it is reserved for future enhancements. The next channel contains 8 bits of the pixel's blue component (256 possible values, from 0 to 255, for the blue component of the pixel's color). The other two channels hold corresponding information for the green and red components of the pixel's color. The three components are used to index the red, green, and blue parts of the look-up table. The RGB (Red, Green, Blue) components from the look-up table combine to produce a pixel with a particular hue and intensity.

Figure 3-4 *XBGR Layout*

31 — 24	23 — 16	15 — 8	7 — 0
unused	blue component	green component	red component

Inverted Video Pixrects

```
pr_blackonwhite(pr, min, max)
Pixrect *pr;
int min, max;

pr_whiteonblack(pr, min, max)
Pixrect *pr;
int min, max;

pr_reversevideo(pr, min, max)
Pixrect *pr;
int min, max;
```

Video inversion is accomplished by manipulation of the colormap of a pixrect. The colormap of a monochrome pixrect has two elements. The procedures `pr_blackonwhite`, `pr_whiteonblack`, and `pr_reversevideo` provide video inversion control. These procedures are ignored for memory pixrects.

In each procedure, `pr` identifies the pixrect to be affected; `min` is the lowest index in the colormap, specifying the background color, and `max` is the highest index, specifying the foreground color. This is often 0 and 1 for monochrome pixrects. The more general definitions allow colormap sharing schemes.

''Black-on-white'' means that zero (background) pixels are painted at full intensity, which is usually white. `pr_blackonwhite()` sets all bits in the entry for colormap location `min` and clears all bits in colormap location `max`.

''White-on-black'' means that zero (background) pixels are painted at minimum intensity, which is usually black. `pr_whiteonblack()` clears all bits in colormap location `min` and sets all bits in the entry for colormap location `max`.

`pr_reversevideo()` exchanges the `min` and `max` color intensities.

NOTE *These procedures are intended for global foreground/background control, not for local highlighting. For monochrome frame buffers, all operations performed after a* `pr_reversevideo()` *call have inverted intensities. For color frame buffers, the behavior is different. The frame buffer's colormap is modified immediately, which affects everything in the display.*

3.8. Attributes for Bitplane Control

In a color pixrect, it is often useful to define bitplanes that can be manipulated independently; operations on one plane leave the other planes of an image unaffected. This is normally done by assigning a plane to a constant bit position in each pixel. Thus, the value of the i^{th} bit in all the pixels defines the i^{th} bitplane in the image. It is sometimes beneficial to restrict pixrect operations to affect a subset of a pixrect's bitplanes. This is done with a bitplane mask. A bitplane mask value is stored in the pixrect's private data and may be accessed by the attribute operations.

Get Plane Mask Attributes

```
#define pr_getattributes(pr, planes)
Pixrect *pr;
int *planes;

#define prs_getattributes(pr, planes)
Pixrect *pr;
int *planes;
```

The macros `pr_getattributes()` and `prs_getattributes()` invoke device-dependent procedures that retrieve the mask controlling which planes in a pixrect are affected by other pixrect operations. `pr` identifies the pixrect; its current bitplanes mask is stored into the word addressed by `planes`. If `planes` is NULL, no operation is performed.

The two macros are identically defined; both are provided to allow consistent use of the same style of names.

Put Plane Mask Attributes

```
#define pr_putattributes(pr, planes)
Pixrect *pr;
int *planes;

#define prs_putattributes(pr, planes)
Pixrect *pr;
int *planes;
```

The macros `pr_putattributes()` and `prs_putattributes()` invoke device-dependent procedures that manipulate a mask controlling which planes in a pixrect are affected by other pixrect operations. The two macros are identically defined; both are provided to allow consistent use of the same style of names.

`pr` identifies the pixrect to be affected. The `planes` argument is a pointer to a bitplane write-enable mask. Only those planes corresponding to mask bits with a value of 1 are affected by subsequent pixrect operations. If `planes` is NULL, no operation is performed.

Note: If any `planes` are masked off by a call to `pr_putattributes()`, no further write access to those planes is possible until a subsequent call to `pr_putattributes()` unmasks them. However, these planes can still be read.

3.9. Plane Groups

A *plane group* is a subset of a frame buffer pixrect. Each plane group is a collection of one or more related bitplanes with stored state (plane mask, colormap, etc.). Each pixrect has a current plane group that is the target of attribute, colormap, and rendering operations.

A plane group is described by a small constant in the header file `<pixrect/pr_planegroups.h>`:

```
#define PIXPG_CURRENT               0
#define PIXPG_MONO                  1
#define PIXPG_8BIT_COLOR            2
#define PIXPG_OVERLAY_ENABLE        3
#define PIXPG_OVERLAY               4
#define PIXPG_24BIT_COLOR           5
#define PIXPG_VIDEO                 6
#define PIXPG_VIDEO_ENABLE          7
#define PIXPG_TRANSPARENT_OVERLAY   8
#define PIXPG_INVALID               127
```

Plane group 0 is the currently active plane group for the pixrect.

A plane group is encoded as a 7-bit field in the pixrect attribute word.

24-Bit Frame Buffers

The CG4, CG8, and the CG9 all have three plane groups. There is a color plane group, which for the CG8 and the CG9 is 24-bits per pixel, and a monochrome overlay plane group with an associated overlay-enable plane group. The overlay is provided for fast monochrome performance of textual windows.

The CG8 and the CG9 have overlay/overlay-enable implementation enhancements over the CG4. A zero in the CG4 overlay-enable causes the 8-bit plane group value for that pixel, rather than the overlay 1-bit value, to be displayed. The CG8 and CG9 require both the overlay-enable and the overlay planes to be zero in order to show the 24-bit color plane group value. This implementation thereby allows three overlay colors rather than the two available with the CG4. The two implementations are compared in the following table.

Table 3-4 *Enable/Overlay Planes for CG4 and CG8/CG9*

Overlay Plane	*Enable Plane*	*CG4 Scheme*	*CG8/CG9 Scheme*
0	0	8-bit color	24-bit color
0	1	color 0	color 1
1	0	8-bit color	color 2
1	1	color 1	color 3

The 24-bit plane group `PIXPG_24BIT_COLOR` provides 24-bit RGB values stored in XBGR format in 32-bit pixels. (See the next subsection for a discussion of XBGR format.) All of the normal logical operations and plane masking are available.

As shown in the following table, the CG8 and CG9 also have one overlay plane and one overlay-enable plane— a total of three plane groups for CG8 and CG9 pixrects.

NOTE *The CG4's enable plane served as a toggle switch that mediated between the monochrome and 8-bit plane groups. The CG8 and CG9 extend the overlay-enable concept. They treat these planes as a 2-bit deep overlay with its own 2-bit deep colormap.*

Table 3-5 *CG8 & CG9 Plane Groups*

Plane	*Function*
`PIXPG_OVERLAY`	Window System Plane
`PIXPG_OVERLAY_ENABLE`	Window System Plane
`PIXPG_24BIT_COLOR`	24-bit Color Plane

The overlay and enable planes are individually accessed as 1-bit deep frame buffers. For each pixel, if both the overlay and overlay-enable planes are zero, the 24-bit frame buffer is visible. If any of the planes are non-zero, the pixel displays the color indicated in the following table:

Table 3-6 *Enable/Overlay Planes for the CG8 and CG9*

Overlay Plane	*Enable Plane*	*Color Index*
0	0	transparent
0	1	1
1	0	2
1	1	3

The `pr_putcolormap` and `pr_getcolormap` functions behave exactly like the CG4 overlay colormap model. Through the use of the `pr_putlut` and `pr_getlut` macro definitions, the CG8 or CG9 overlay color model is used.

Consider the following examples of the `pr_putcolormap` and `pr_putlut` functions. Logically, `pr_putcolormap` has two entries (monochrome) while `pr_putlut` has four entries (color). After this call is issued:

```
pr_putcolormap(pr, 0, 2, r, g, b)
```

the colors of the overlay planes are as follows. Note that `r`, `g`, and `b` can be any value.

Colormap Index	*Color*
0	transparent
1	r[0] g[0] b[0]
2	unchanged
3	r[1] g[1] b[1]

The values `r[0]`, `g[0]`, and `b[0]` are placed in the colormap index 1. Index 2 remains unchanged, while index 3 contains the values `r[1]`, `g[1]`, and `b[1]`.

After the call:

```
pr_putlut(pr, 0, 4, r, g, b)
```

the colors of the overlay planes are as follows:

Colormap Index	*Color*
0	transparent
1	r[1] g[1] b[1]
2	r[2] g[2] b[2]
3	r[3] g[3] b[3]

NOTE *The CG8 or CG9's default plane group is the overlay plane group, not the 24-bit plane group.*

The following example code shows how to test whether the color board that the application uses supports 24-bit color. This type of code is important for writing portable software that can run with either 8 or 24-bit color.

```
#include <pixrect/pixrect_hs.h>

char maxgroup[PIXPG_24BIT_COLOR + 1];
pr_available_plane_groups(pr, PIXPG_24BIT_COLOR + 1, maxgroup);
if (maxgroup[PIXPG_24BIT_COLOR] != 0)
    printf("Board supports 24-bit color\n");
```

Determine Supported Plane Groups

```
ngroups = pr_available_plane_groups(pr, maxgroups, groups);
Pixrect *pr;
int maxgroups;
char groups[maxgroups]
```

`pr_available_plane_groups` allows you to determine which plane groups are supported by the machine you are working on. `pr_available_plane_groups` fills the character array `groups` with true (`1`) values for the plane groups implemented by the pixrect `pr`. The entry for the current plane group (`groups[0]`) array is always set to false (`0`). The size of `groups` is passed to the function as `maxgroups` to avoid overwriting the end of the array.

`pr_available_plane_groups` returns the index of the highest-numbered implemented plane group, plus one.

Get Current Plane Group

```
group = pr_get_plane_group(pr);
Pixrect *pr;
```

`pr_get_plane_group` returns the current plane group number for the pixrect `pr`. If the current plane group is unknown, the function returns `PIXPG_CURRENT`.

Set Plane Group and Mask

```
void pr_set_plane_group(pr, group);
Pixrect *pr;
int group;

void pr_set_planes(pr, group, planes)
Pixrect *pr;
int group;
int planes;
```

`pr_set_plane_group` sets the current plane group for the pixrect `pr` to the value given by `group`. If this plane group is `PIXPG_CURRENT` or unimplemented, `pr_set_plane_group` does nothing.

The `pr_set_planes` function is equal to a `pr_set_plane_group(pr, group)` followed by `pr_putattributes(pr, &planes)`. `planes` contains a bitplane write-enable mask. Only those planes corresponding to mask bits having a value of `1` are affected by subsequent pixrect operations. However, the other planes can still be read.

3.10. Double Buffering

Some frame buffers have double buffering support implemented in hardware. Two pixrect commands, `pr_dbl_get()`, and `pr_dbl_set()` allow you to inquire about and control a double-buffered display device. The pixrect interface assigns two names to the buffers in the display; `PR_DBL_A` for one, and `PR_DBL_B` for the other.

A buffer can be *displayed*, *read*, or *written*. When a buffer is displayed, its stored image is shown on the screen. If the software requests that the other buffer be displayed, the hardware does not switch to the new buffer until the next vertical retrace of the screen. This prevents any flicker from showing on the screen during the change between buffers. A buffer can be read or written, using pixrect commands, at any time.

Get Double-Buffering Attributes

```
state = pr_dbl_get(pr, attribute)
Pixrect *pr;
int attribute;
```

This function shows the current attributes of the double buffer. You can inquire about the state of the display device by executing `pr_dbl_get` with a particular attribute value, then examining the function's return value. The legal attributes are listed below:

```
#define PR_DBL_AVAIL    1
#define PR_DBL_DISPLAY  2
#define PR_DBL_WRITE    3
#define PR_DBL_READ     4
```

The PR_DBL_AVAIL returns PR_DBL_EXISTS if the display device has hardware double buffering capacity. Otherwise, it returns NULL. The other attributes indicate which buffer on the device is being displayed and which can be written to. The possible state values for these attributes are given below:

```
#define PR_DBL_A    2
#define PR_DBL_B    3
#define PR_DBL_BOTH 4
#define PR_DBL_NONE 5
```

Not all return values are possible with each attribute. The values that can be returned for a given attribute are shown in the table below:

Table 3-7 pr_dbl_get() *Attributes*

Attribute	*Possible Values Returned*
PR_DBL_AVAIL	PR_DBL_EXISTS
PR_DBL_DISPLAY	PR_DBL_A, PR_DBL_B
PR_DBL_WRITE	PR_DBL_A, PR_DBL_B, PR_DBL_BOTH, PR_DBL_NONE
PR_DBL_READ	PR_DBL_A, PR_DBL_B

Set Double-Buffering Attributes

```
void pr_dbl_set(pr, attribute_list)
Pixrect *pr;
int *attribute_list;
```

The pr_dbl_set() function changes the state of the double buffering display. It controls the buffer being displayed, and selects the buffer(s) affected by pixrect reads and writes. The possible attributes for pr_dbl_set() are given below:

```
#define PR_DBL_DISPLAY              2
#define PR_DBL_WRITE                3
#define PR_DBL_READ                 4
#define PR_DBL_DISPLAY_DONTBLOCK    5
```

An attribute list is an integer array containing attributes/value pairs. The last element of the array should be zero. If the display is already in the state requested, the function simply returns.

If the PR_DBL_DISPLAY attribute is in the list, then the function may block up to a single video frame's time (15 ms) waiting for the next vertical retrace. This action ensures that the next pixrect operation does not alter the buffer while it is still being displayed. Applications that do not write to the buffer for at least 15 ms after changing the displayed buffer, and that need maximum throughput, can use PR_DBL_DISPLAY_DONTBLOCK. This attribute changes the display without blocking the process until the next vertical retrace.

NOTE *Programmers should use* PR_DBL_DISPLAY_DONTBLOCK *with caution. If the application starts writing too early, this action modifies the buffer while it is still being displayed.*

The definitions of all the possible attribute values are shown below:

```
#define PR_DBL_A    2
#define PR_DBL_B    3
#define PR_DBL_BOTH 4
```

Not all of the values can be paired with all of the attributes; the allowed pairings are shown in the table below:

Table 3-8 `pr_dbl_set()` *Attributes*

Attribute	*Possible Values to Set*
PR_DBL_WRITE	PR_DBL_A, PR_DBL_B, PR_DBL_BOTH
PR_DBL_READ	PR_DBL_A, PR_DBL_B
PR_DBL_DISPLAY_DONTBLOCK	PR_DBL_A, PR_DBL_B
PR_DBL_DISPLAY	PR_DBL_A, PR_DBL_B

On the **CG9** *true color* frame buffer, the `PR_DBL_WRITE` attribute also controls double buffering. These calls and the modes that they enable are summed up in the table below:

Table 3-9 *24-Bit True Color Double Buffering*

Pixrect Call	Buffering Mode Enabled
pr_dbl_set (*Pixrect, PR_DBL_WRITE, PR_DBL_A)	12-bit Double Buffering
pr_dbl_set (*Pixrect, PR_DBL_WRITE, PR_DBL_B)	12-bit Double Buffering
pr_dbl_set (*Pixrect, PR_DBL_WRITE, PR_DBL_BOTH)	24-bit True Color

Note that setting the **CG9** to write to both buffers is the means for returning to 24-bit mode.

3.11. Efficiency Considerations

For maximum execution speed, remember the following points when you write pixrect programs:

- `pr_get` and `pr_put()` are relatively slow. For fast random access of pixels, it is usually faster to read an area into a memory pixrect and address the pixels directly.
- `pr_rop()` is fast for large rectangles.
- `pr_vector()` is fast.
- Functions run faster when clipping is turned off. Do this only if you can guarantee that all accesses are within the pixrect bounds.

- `pr_rop()` is three to five times faster than `pr_stencil()`.
- `pr_batchrop()` cuts down the overhead of painting many small pixrects.
- For small standard shapes `pr_rop()` should be used instead of `pr_polygon_2()`.
- `pr_polyline()` is an efficient way to draw a series of vectors.
- `pr_polypoint()` is faster than a series of `pr_puts()` or single pixel `pr_rops()`. It is useful for implementing new primitives such as curves.
- The `PR_DBL_DISPLAY_DONTBLOCK` attribute of `pr_dbl_set()`, if used appropriately, can speed up animation sequences.

4

Text Facilities for Pixrects

4

Text Facilities for Pixrects

The Pixrect Library contains higher-level facilities for displaying text. These facilities fall into two main categories: a standard format for describing fonts and character images, including routines for processing them; and a set of routines that take a string of text and a font, and handle various parts of painting that string in a pixrect.

4.1. Pixfonts and Pixchars

```
struct pixchar {
    struct pixrect *pc_pr;
    struct pr_pos pc_home;
    struct pr_pos pc_adv;
};
```

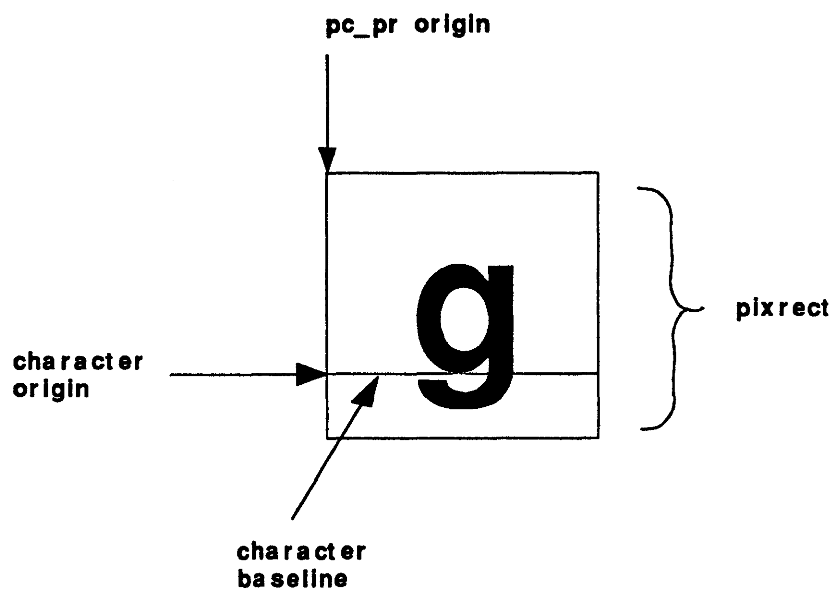
The `pixchar` structure defines the format of a single character in a font. The actual image of the character is a pixrect (a separate pixrect for each character) addressed by `pc_pr`. The entire pixrect gets painted. Characters that do not have a displayable image have NULL in their `pc_pr` entry. `pc_home` is the origin of pixrect `pc_pr` (its upper-left corner) relative to the character origin. A character's origin is the left-most end of its *baseline*, the lowest point on characters without descenders. Figure 4-1 illustrates the `pc_pr` origin and the character origin.

The left-most point on a character is normally its origin, but *kerning* or mandatory letter spacing may move the origin to the right or left of that point. `pc_adv` is the amount the destination position is changed by this character; that is, the amounts in `pc_adv` added to the current character origin will give the origin for the next character. While normal text advances only horizontally, rotated fonts may have a vertical advance. Both are provided for in the font.

```
typedef struct pixfont {
    struct pr_size pf_defaultsize;
    struct pixchar pf_char[256];
} Pixfont;
```

The `Pixfont` structure contains an array of `pixchars`, indexed by the character code; it also contains the size (in pixels) of its characters when they are all the same. If the size of a font's characters varies in one dimension, that value in `pf_defaultsize` will not have anything useful in it; however, the other may still be useful. Thus, for non-rotated variable-pitch fonts, `pf_defaultsize.y` will still indicate the unleaded interline spacing for that font.

Figure 4-1 *Character and* `pc_pr` *Origins*



4.2. Operations on Pixfonts

The commands listed below allow you to load a font to display. A font must be loaded before any text operation can be performed.

Load a Font

```
Pixfont *pf_open(name)
char *name;
```

`pf_open()` returns a pointer to a *shared* copy of a font in virtual memory. A NULL is returned if the font cannot be opened. The path name of the font file should be specified. For example:

```
myfont = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.7");
```

`name` should be in the format described in *vfont*(5): the file is converted to pixfont format, allocating memory for its associated structures and reading in the data for it from disk. The utility `fontedit(1)` is a font editor for designing pixel fonts in *vfont*(5) format.

The `pf_open()` routine sets the `pf_defaultsize` values of a new pixfont by using the following criteria:

The default width, `pf_defaultsize.x`, is the width (in pixels) of the font's lower case "a," if one exists in the font. The default interline spacing, `pf_defaultsize.y`, is 1-1/2 the height, in pixels, of the font's upper case "a" (A), measured from the font baseline.

The data from a small selection of commonly used fonts is compiled into the Pixrect Library. The names of these built-in fonts are checked against the last component of the name. To guarantee that the font is loaded from the disk file instead, use `pf_open_private()` instead of `pf_open()`.

Load Private Copy of Font

```
Pixfont *pf_open_private(name)
char *name;
```

`pf_open_private()` returns a pointer to a private copy of a font in virtual memory. A NULL is returned if the font cannot be opened.

Default Fonts

```
Pixfont *pf_default()
```

The procedure `pf_default` performs the same open function for the system default font, normally a fixed-pitch, 16-point sans serif font with upper-case letters 12 pixels high. If the environment parameter DEFAULT_FONT is set, its value will be taken as the name of the font file to be opened by `pf_default()`.

Close Font

```
pf_close(pf)
Pixfont *pf;
```

When a client is finished with a font, it should call `pf_close()` to free the memory associated with it. `pf` should be a font handle returned by a previous call to `pf_open()`, `pf_open_private()` or `pf_default()`.

4.3. Text Functions

The following functions manage various tasks involved in displaying text.

Pixrect Text Display

```
pf_text(where, op, font, text)
struct pr_prpos where;
int op;
Pixfont *font;
char *text;
```

Characters are written into a pixrect with the `pf_text()` procedure. `where` is the destination for the start of the text (nominal left edge, baseline; see Section 4.1). `op` is the raster operation to be used in writing the text, as described in Section 3.3, *The Op Argument*. `font` is a pointer to the font in which the text is to be displayed. `text` is the actual null-terminated string to be displayed. The color specified in the `op` specifies the color of the ink. The background of the text is painted 0 (background color).

Transparent Text

```
pf_ttext(where, op, font, text)
struct pr_prpos where;
int op;
Pixfont *font;
char *text;
```

pf_ttext paints transparent text. It does not disturb destination pixels in blank areas of the character's image. The arguments to this procedure are the same as for pf_text(). The character's bitmap is used as a stencil, and the color specified in op is painted through the stencil.

For monochrome pixrects, the same effect can be achieved by using PIX_SRC | PIX_DST as the function in the op; this procedure is for color pixrects.

Auxiliary Pixfont Procedures

```
struct pr_size pf_textbatch(where, lengthp, font, text)
struct pr_prpos where[];
int *lengthp;
Pixfont *font;
char *text;

struct pr_size pf_textwidth(len, font, text)
int len;
Pixfont *font;
char *text;
```

pf_textbatch() is used internally by pf_text(). It constructs an array of pr_pos structures and records its length, as required by batchrop (see Section 3.6). where should be the address of the array to be filled in, and lengthp should point to a maximum length for that array. text addresses the null-terminated string to be put in the batch, and font refers to the Pixfont that displays it. When the function returns, lengthp refers to a word containing the number of pr_pos structures actually used for text. The pr_size returned is the sum of the pc_adv fields in their pixchar structures.

pf_textwidth() returns a pr_size that is computed by taking the product of len (the number of characters), and pc_adv, (the width of each character).

Text Bounding Box

```
pf_textbound(bound, len, font, text)
struct pr_subregion *bound;
int len;
Pixfont *font;
char *text;
```

pf_textbound may be used to find the bounding box for a string of characters in a given font. bound->pos is the top-left corner of the bounding box, bound->size.x is the width, and bound->size.y is the height. bound->pr is not modified. bound->pos is computed relative to the location of the character origin (base point) of the first character in the text.

Unstructured Text

```
pr_text(pr, x, y, op, font, text)
Pixrect *pr;
int x, y, op;
Pixfont *font;
char *text;

pr_ttext(pr, x, y, op, font, text)
Pixrect *pr;
int x, y, op;
Pixfont *font;
char *text;
```

These unstructured text functions correspond to the Pixwin functions `pw_text()` and `pw_ttext()`. `prs_text()` and `prs_ttext()` macros are also provided, although they are identical to `pf_text()` and `pf_ttext()`, respectively.

4.4. Example

Here is an example program that writes text on the display surface with pixel fonts.

Figure 4-2 *Example Program Using Text*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *pr;
    Pixfont *pf;

    if (!(pr = pr_open("/dev/fb")) ||
        !(pf = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.12")))
        exit(1);

    pr_text(pr, 400, 400, PIX_SRC, pf, "This is a string.");

    pr_close(pr);
    pf_close(pf);
    exit(0);
}
```

5

Memory Pixrects

5

Memory Pixrects

Memory pixrects store their pixels in the system memory, instead of displaying them. They are similar to other pixrects but have several special properties. Like all other pixrects, their dimensions are visible in the `pr_size` and `pr_depth` elements of their pixrect structure. The device-dependent operations used to manipulate them are available through their `pr_ops` structure pointer. However, the format of the data that describes the particular pixrect is also public: `pr_data` holds the address of an `mpr_data` structure described below. Therefore, a client program may construct and manipulate memory pixrects using non-pixrect operations. There is also a function `mem_create()` that dynamically allocates a new memory pixrect and a macro `mpr_static()`, that can be used to generate an initialized memory pixrect in the code of a client program.

5.1. The `mpr_data` Structure

```
struct mpr_data {
    int md_linebytes;
    short *md_image;
    struct pr_pos md_offset;
    short md_primary;
    short md_flags;
};

/* md_flags bit definitions */
#define MP_REVERSEVIDEO 1
#define MP_DISPLAY      2
#define MP_PLANEMASK    4
#define MP_I386         8
#define MP_STATIC       16
```

The `pr_data` member of a memory pixrect points to an `mpr_data` structure, which contains the information needed to access a memory pixrect.

`md_linebytes` is the number of bytes stored in each row of the primary pixrect. This is the difference in the addresses of two pixels at the same x-coordinate, one row apart. Since a secondary pixrect may not include the full width of its primary pixrect and the amount of padding at the end of a scan line may vary, this quantity cannot be computed from the width of the pixrect — see Section 3.4.

The actual pixels of a memory pixrect are stored in an array to which md_image points. (The format of this area is described in a later section). The creator of the memory pixrect must ensure that md_image contains a 16-bit aligned address; a 32-bit aligned address is required for 32-bit deep memory pixrects and is recommended in all cases for best performance.

md_offset is the (x, y) position of the first pixel of this pixrect in the array of pixels addressed by md_image. Both values will be zero for a primary pixrects.

md_primary is 1 if the pixrect is primary and if its image was allocated dynamically (that is, by mem_create()). In this case, md_image points to an area not referred to by any other primary pixrect. If this flag is set, the pixrect's image memory is freed when the pixrect is destroyed by pr_destroy().

The MP_DISPLAY bit is set in md_flags if the memory pixrect is actually a memory mapped frame buffer. The MP_REVERSEVIDEO bit is set if reverse video is currently in effect for the pixrect. (This is only valid if the pixrect is 1 bit deep). The MP_PLANEMASK bit is set if the memory pixrect private data is actually a mprp_data structure, which stores a bit plane mask. These flags are used to support memory-mapped display devices, such as the bwtwo and cgthree frame buffers.

The MP_386I bit is set if the pixrect image data is in 80386 format (leftmost pixel in the least significant bits). The MP_STATIC bit is non-zero if the pixrect is static. These two flags are used to determine if bit flipping is necessary to display the pixrect on a Sun386i machine. See Chapter 2 for details on 80386 format, and the MP_386I and MP_STATIC flags.

NOTE *The* MP_386I *and* MP_STATIC *flags are ignored on SPARC and 680X0 machines.*

Several useful macros are defined in <pixrect/memvar.h>. Three commonly used macros are described here; see the others in memvar.h.

Use the mpr_d() macro to access a memory pixrect's bitmap. It generates a pointer to the private data of a memory pixrect:

```
#define mpr_d(pr)    ((struct mpr_data *) (pr)->pr_data)
```

The mpr_linebytes() macro computes the bytes per line of a 16-bit padded primary memory pixrect given its width in pixels and the bits per pixel:

```
#define mpr_linebytes(width, depth)
    ( ((pr_product(width, depth)+15)>>3) &~1)
```

It is useful for computing the amount of space required for a static pixrect or an image data array which is to be passed to mem_point(). However, mpr_linebytes() should not be used to access the image data of an existing memory pixrect. To examine image data use md_linebytes directly, or the mpr_mdlinebytes() macro:

```
#define mpr_mdlinebytes(mpr)     (mpr_d(mpr)->md_linebytes)
```

Example

An example program that uses a memory pixrect to perform bit manipulations on the screen follows. It opens the frame buffer and copies the bitmap to a memory pixrect of the same size. It then goes through each byte of the memory pixrect, left-shifting each byte (this is not a useful operation, just a simple example). Finally, it copies the modified memory pixrect to the screen pixrect.

Note how `md_linebytes` is multiplied by the pixrect height to find the total size of the memory pixrect image data array.

Figure 5-1 *Example Program Using Memory Pixrects*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *scrn, *mem;
    int w, h;
    char *start, *end, *ptr;

    if ((scrn = pr_open("/dev/fb") == 0)
        exit(1);
    w = scrn->pr_size.x;
    h = scrn->pr_size.y;
    if ((mem = mem_create(w, h, scrn->pr_depth)) == 0)
        exit(1);
    (void) pr_rop(mem, 0, 0, w, h, PIX_SRC, scrn, 0, 0);
    start = (char *) mpr_d(mem)->md_image;
    end = start + h * mpr_d(mem)->md_linebytes;
    for (ptr = start; ptr < end; ptr++)
        *ptr <<= 2;
    (void) pr_rop(scrn, 0, 0, w, h, PIX_SRC, mem, 0, 0);
    (void) pr_close(mem);
    (void) pr_close(scrn);
    exit(0);
}
```

5.2. Creating Memory Pixrects

The `mem_create()` and `mem_point()` functions allow a client program to create memory pixrects.

Create Memory Pixrect

```
Pixrect *mem_create(w, h, depth)
int w, h, depth;
```

A new primary pixrect is created by a call to the function `mem_create()`. `w`, `h`, and `depth` specify the width and height in pixels, and depth in bits per pixel of the new pixrect. Sufficient memory to hold those pixels is allocated and cleared to 0. New `mpr_data` and pixrect structures are allocated and initialized, while a pointer to the pixrect is returned. If this cannot be done (usually because of insufficient swap space), the return value is 0.

On 32-bit machines, such as the Sun-3, Sun-4, and Sun386i, the created pixrect has each scan line padded out to a 32-bit boundary, unless it is only 16 bits wide; that is, the `md_linebytes` structure member contains either 2 or a multiple of 4. On Sun-3 workstations, the SunOS releases prior to 4.0, pixrects created by `mem_create()` were always padded to a 16-bit boundary.

On Sun386i machines, the memory pixrects created by `mem_create()` have the `MP_I386` flag set.

Create Memory Pixrect from an Image

```
Pixrect *mem_point(width, height, depth, data)
int width, height, depth;
short *data;
```

The `mem_point()` function builds a pixrect structure that points to a dynamically created image in memory. Client programs may use this function as an alternative to `mem_create()` if the image data is already in memory. `width` and `height` are the width and height of the new pixrect, in pixels. `depth` is the depth of the new pixrect, in number of bits per pixel. `data` points to the image data to be associated with the pixrect.

Note that `mem_point()` expects each line of the memory image to be padded to a 16-bit boundary. If the image data has greater padding (32-bit padding is recommended), `md_linebytes` should be set to the correct value after calling `mem_point()`. Also, `mem_point()` does not set the `md_primary` flag, so the image data is not automatically freed when the pixrect is destroyed.

On Sun386i machines, the `mem_point()` function does not set the `MP_386I` flag. The image data supplied to `mem_point()` should be in SPARC/680X0 format (leftmost pixel in the most significant bits).

Example

Here is an example program that uses a memory pixrect to invert the frame buffer contents from top to bottom. It opens the default frame buffer and creates a memory pixrect of the same size. It then copies rows of pixels from the frame buffer to the memory pixrect in reverse order. Finally, it copies the memory pixrect to the frame buffer.

Figure 5-2 *Example Program Using Memory Pixrects*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *pr, *tmp;
    int yin, yout;

    if (!(pr = pr_open("/dev/fb")) ||
        !(tmp =
        mem_create(pr->pr_size.x, pr->pr_size.y, pr->pr_depth)))
        exit(1);

    for (yin = 0, yout = pr->pr_size.y - 1; yout >= 0; yin++, yout--)
        pr_rop(tmp, 0, yout, pr->pr_size.x, 1, PIX_SRC, pr, 0, yin);

    pr_rop(pr, 0, 0, pr->pr_size.x, pr->pr_size.y, PIX_SRC, tmp, 0, 0);

    exit(0);
}
```

5.3. Static Memory Pixrects

```
#define mpr_static(name, w, h, depth, image)
int w, h, depth;
short *image;
```

A memory pixrect may be created at compile time by using the `mpr_static()` macro. `name` is a unique token to identify the generated data objects; `w`, `h`, and `depth` are the width and height in pixels, and depth in bits of the pixrect; and `image` is the address of a 16-bit aligned (32-bit aligned if `depth` is 32) data object that contains the pixel values in the format described below, with each line padded to a 16-bit boundary.

The macro generates two structures:

```
struct mpr_data name_data;
Pixrect name;
```

The `mpr_data` structure is initialized to point to image data specified. The pixrect structure is initialized with `mem_ops` and `name_data`.

On a Sun386i machine, the `MP_STATIC` flag is set in the md_flags byte of the pixrect data structure; see Chapter 2 for details.

NOTE *Contrary to its name, this macro generates structures of storage class* `extern`. *The* `mpr_static_static()` *macro accepts the same arguments as* `mpr_static()`, *but generates* `static` *structure declarations.*

5.4. Pixel Layout in Memory Pixrects

In memory, the upper-left corner pixel is stored in the word at the lowest address. This address must be 16-bit aligned (32-bit aligned for 32-bit deep pixrects). The first word is followed by words containing the remaining pixels in the top row, left-to-right. Pixels are stored in successive bits without padding or alignment.

The order of pixels within each word is determined by the machine architecture. On SPARC and 680X0 machines, the leftmost pixel is stored in the most significant bits of the word, while on 80386 machines the preferred order is to store the leftmost pixel in the least significant bits of the word.

Each row of pixels is rounded to at least a 16-bit boundary. For best performance on 32-bit machines, pixel rows should be rounded to 32-bit boundaries (mem_create does this automatically). However, 16-bit rounding is required for static pixrects and mem_point.

NOTE *On Sun386i machines, a pixrect's image data is converted to 80386 format before being displayed. See Chapter 2 for details.*

Memory pixrects with depths of 1, 8, 16, and 32 bits are fully supported by the Pixrect Library.

You can create memory pixrects with other depths (such as 24 bits) and write them to raster files with `pr_dump()`, but none of the pixrect drawing functions can be used on them. The `pr_load()` function automatically converts 24-bit raster files to 32-bit memory pixrects when the files are read.

5.5. Using Memory Pixrects

Memory pixrects can be used to read data from and write data to frame buffers. Several functions exist for interfacing Pixwins with memory pixrects. These include `pw_read()`, `pw_rop()` and `pw_write()`. Refer to the *SunView 1 Programmer's Guide* for more details. For applications using a raw frame buffer device without SunView, `pr_rop()` can be used for operations on memory pixrects. Another use of memory pixrects is the processing of images not intended for display. User programs can write directly into a pixrect using parameters found in the `mpr_data` structure, or they can use `mem_point()` for a previously created image.

Memory pixrects can also be written to raster files using the facilities described in Chapter 6.

6

File I/O Facilities for Pixrects

6

File I/O Facilities for Pixrects

Sun Microsystems, Inc. has specified a file format for files containing raster images. The format is defined in the header file `<rasterfile.h>`. The Pixrect Library contains routines to perform I/O operations between pixrects and files in this raster file format. This I/O is done using the routines of the C Library Standard I/O package, requiring the caller to include the header file `<stdio.h>`.

The raster file format allows multiple types of raster images. Unencoded, and run-length encoded formats are supported directly by the Pixrect Library. Support for customer-defined formats is implemented by passing raster files with non-standard types through filter programs. Sun-supplied filters are found in the directory `/usr/lib/rasfilters`. This directory also includes sample source code for a filter that corresponds to one of the standard raster file types to facilitate writing new filters.

6.1. Writing and Reading Raster Files

The sections that follow describe how to store and retrieve an image in a raster file.

Run Length Encoding

The run-length encoding used in raster files is of the form

```
<byte><byte>...<ESC><0>...<byte><ESC><count><byte>...
```

where the `count` value can range from 0 to 255. This value indicates that following `byte` should appear `count` + 1 times in the actual image. This means the count/byte pair can represent 1 to 256 consecutive instances of `byte` in the image. One or two byte sequences are left unencoded; only sequences of three or more of the same byte value are encoded as `<ESC><count><byte>`. A byte with the value `<ESC>` indicates that the next two bytes should be interpreted as a count/byte pair. The integer value of the escape byte is 128. To represent the value 128 (`<ESC>`), each instance is encoded as `<ESC><0>`, since the `<count>` in this scheme can never be 0, since single bytes are not encoded with count/byte pairs. The byte position of a count/byte pair can be any eight bit values; a pair of 128 values, `<ESC><ESC>` is encoded as `<ESC><1><ESC>`.

This algorithm fails only if the input stream contains an excessive number of one and two-character sequences of the `<ESC>` character. Such an image can be translated successfully, and will faithfully represent the original bitmap, but the "compressed" image is larger that the original one!

Write Raster File

```
int pr_dump(input_pr, output, colormap, type, copy_flag)
Pixrect *input_pr;
FILE *output;
colormap_t *colormap;
int type, copy_flag;
```

The `pr_dump()` procedure stores the image described by a pixrect onto a file. It normally returns 0, but if any error occurs, it returns PIX_ERR. The caller can write a rectangular sub-region of a pixrect by first creating an appropriate `input_pr` via a call to `pr_region()`. The output file is specified via `output`. The specified output type should either be one of the following standard types, or should correspond to a customer-provided filter.

```
#define RT_OLD      0
#define RT_STANDARD 1
#define RT_BYTE_ENCODED 2
```

The RT_STANDARD type is the common raster file format in the same sense that memory pixrects are the common pixrect format: every raster file filter is required to read and write this format. The RT_OLD type is very close to the RT_STANDARD type; it was the former standard generated by old versions of Sun software. The RT_BYTE_ENCODED type implements a run-length byte-encoding of the pixrect image. This usually results in shorter files, although the worst case could cause the image to expand up to 50 percent.

Specifying any other output type causes `pr_dump()` to pipe a raster file of RT_STANDARD type to the filter named `convert.`*type*. Where *type* is the ASCII string corresponding to the decimal value of the type.

It looks for this filter first in directories in the user's `$PATH` environment variable, and then in the directory `/usr/lib/rasfilters`. The output of the filter is then copied to `output`.

It is strongly recommended that customer-defined formats use a type value of 100 or more, to avoid conflicts with additions to the set of standard types. The RT_EXPERIMENTAL type is reserved for use in the development of experimental filters, although it is no longer treated specially.

```
#define RT_EXPERIMENTAL 65535
```

`pr_dump()` and other functions that start filters wait until the filter process exits before returning, so caution is advisable when you are working with experimental filters.

For pixrects displayed on devices with colormaps, the values of the pixels are not sufficient to recreate the displayed image. Thus, the image's colormap can also be specified in the call to `pr_dump()`. If the `colormap` is specified as NULL, but `input_pr` is a non-monochrome display pixrect, `pr_dump()` attempts to write the colormap obtained from `input_pr` (via `pr_getcolormap`). The following structure specifies the colormap associated with `input_pr`:

```
typedef struct {
    int type;
    int length;
    unsigned char *map[3];
} colormap_t;
```

The colormap type should be one of the Sun-supported types:

```
#define RMT_NONE 0
#define RMT_EQUAL_RGB 1
#define RMT_RAW 2
```

If the colormap type is RMT_NONE, then the colormap length must be 0. This case usually arises when you are dealing with monochrome displays and 1-bit deep memory pixrects. If the colormap type is RMT_EQUAL_RGB, then the map array should specify the red (`map[0]`), green (`map[1]`) and blue (`map[2]`) colormap values, with each vector in the map array being of the same specified colormap length. If the colormap type is RMT_RAW, the first map array (`map[0]`), should hold `length` bytes of colormap data, which is not interpreted by the Pixrect Library.

Finally, `copy_flag` specifies whether or not `input_pr` should be copied to a temporary pixrect before the image is output. The `copy_flag` value should be non-zero if `input_pr` is a pixrect in a frame buffer that is likely to be asynchronously modified. The copy flag is also automatically set to a non-zero value for secondary pixrects, to simplify the code. Note that use of `copy_flag` still does not guarantee that the correct image is output unless the `pr_rop()` to copy from the frame buffer is made uninterruptible.

Figure 6-1 *Example Program using* `pr_dump()`

```
#include <stdio.h>
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_io.h>

main()
{
    Pixrect *screen, *icon;
    FILE *output = stdout;
    colormap_t *colormap = 0;
    int type = RT_STANDARD;
    int copy_flag = 1;

    if (!(screen = pr_open("/dev/fb")) ||
        !(icon = pr_region(screen, 1050, 10, 64, 64)))
        exit(1);

    pr_dump(icon, output, colormap, type, copy_flag);
    pr_close(screen);

    exit(0);
}
```

Read Raster File

```
Pixrect *pr_load(input, colormap)
FILE *input;
colormap_t *colormap;
```

The `pr_load()` function can be used to retrieve the image stored in a raster file into a pixrect. The raster file's header is read from `input`, a pixrect of the appropriate size is dynamically allocated, the colormap is read and placed in the location addressed by `colormap`, and finally the image is read into the pixrect and the pixrect returned. If any problem occurs, `pr_load()` returns NULL. Note that 24-bit raster files are loaded as 32-bit pixrect.

As with `pr_dump()`, if the specified raster file is not of standard type, `pr_load()` first runs the file through the appropriate filter to convert it to RT_STANDARD type and then loads the output of the filter.

Additionally, if `colormap` is NULL, `pr_load()` simply discards any and all colormap information contained in the specified input raster file. If `colormap` is non-null, `pr_load()` loads the colormap data even if the type and length specified do not match that of the file (see `pr_load_colormap()` below).

Figure 6-2 *Example Program using* `pr_load()`

```
#include <stdio.h>
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_io.h>

main()
{
    Pixrect *screen, *icon;
    FILE *input = stdin;
    colormap_t colormap;

    colormap.type = RMT_NONE;

    if (!(screen = pr_open("/dev/fb")) ||
        !(icon = pr_load(input, &colormap)))
        exit(1);

    if (colormap.type == RMT_EQUAL_RGB)
        pr_putcolormap(screen, 0, colormap.length,
            colormap.map[0], colormap.map[1],
            colormap.map[2]);

    pr_rop(screen, 1050, 110, 64, 64, PIX_SRC, icon, 0, 0);
    pr_close(screen);

    exit(0);
}
```

6.2. Details of the Raster File Format

A handful of additional routines are available in the Pixrect Library for manipulating pieces of raster files. In order to understand what they do, it is necessary to understand the exact layout of the raster file format.

The raster file is in three parts: first, a small header containing eight 32-bit `int`'s; second, a (possibly empty) set of colormap values; third, the pixel image, stored a line at a time, in increasing y order.

The image is essentially laid out in the file the exact way that it would appear in a static memory pixrect. In particular, each line of the image is rounded out to a multiple of 16 bits, corresponding to the rounding convention used by static pixrects.

The header is defined by the following structure:

```
struct rasterfile {
    int ras_magic;
    int ras_width;
    int ras_height;
    int ras_depth;
    int ras_length;
    int ras_type;
    int ras_maptype;
    int ras_maplength;
};
```

The `ras_magic` field always contains the following constant:

```
#define RAS_MAGIC 0x59a66a95
```

The `ras_width`, `ras_height` and `ras_depth` fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is usually either 1 or 8, corresponding to the standard frame buffer depths.

The `ras_length` field contains the length in bytes of the image data. For an unencoded image, this number is computable from the `ras_width`, `ras_height`, and `ras_depth` fields, making `ras_length` redundant, but for an encoded image the value is necessary so the image length is available without having to decode the image itself.

NOTE *The length of the header and of the possibly empty colormap values are* **not** *included in the value in the* `ras_length` *field. The field value is only the length of the image data.*

For historical reasons, files of type RT_OLD usually have a 0 in the `ras_length` field, and software expecting to encounter such files should be prepared to compute the actual image data length if it is needed. The `ras_maptype` and `ras_maplength` fields contain the type and length in bytes of the colormap values, respectively.

If the `ras_maptype` is not RMT_NONE and the `ras_maplength` is not 0, then the colormap values are the `ras_maplength` bytes immediately after the header. These values are either uninterpreted bytes (usually with the `ras_maptype` set to RMT_RAW) or the equal length red, green and blue vectors, in that order (when the `ras_maptype` is RMT_EQUAL_RGB). In the latter case, the `ras_maplength` must be three times the size in bytes of any one of the vectors.

6.3. Writing Parts of a Raster File

The following routines are available for writing the various parts of a raster file. Many of these routines are used to implement `pr_dump()`. First, the raster file header and the colormap can be written by calling `pr_dump_header()`.

Write Header to Raster File

```
int pr_dump_header(output, rh, colormap)
FILE *output;
struct rasterfile *rh;
colormap_t *colormap;
```

`pr_dump_header()` returns PIX_ERR if there is a problem writing the header or the colormap; otherwise it returns 0. If the colormap is NULL, no colormap values are written.

Initialize Raster File Header

```
Pixrect *pr_dump_init(input_pr, rh, colormap,
    type, copy_flag)
Pixrect *input_pr;
struct rasterfile *rh;
colormap_t *colormap;
int type, copy_flag;
```

For clients who do not want to explicitly initialize the raster file struct, this routine can be used to set up the arguments for `pr_dump_header()`. The arguments to `pr_dump_init()` correspond to the arguments to `pr_dump()`. However, `pr_dump_init()` returns the pixrect to write, rather than actually writing it, and initializes the structure pointed to by `rh` rather than writing it. If colormap is NULL, the `ras_maptype` and `ras_maplength` fields of `rh` are set to RMT_NONE and 0, respectively.

If any error is detected by `pr_dump_init()`, the returned pixrect is NULL. If there is no error, and the `copy_flag` is zero, then the input pixrect is suitable for direct dumping (it is a primary memory pixrect). The returned pixrect is simply `input_pr`. However, if `copy_flag` is non-zero, or the input pixrect cannot be dumped directly, the returned pixrect is dynamically allocated and the caller is responsible for deallocating it with `pr_destroy()` when it is no longer needed.

Write Image Data to Raster File

```
int pr_dump_image(pr, output, rh)
Pixrect *pr;
FILE *output;
struct rasterfile *rh;
```

The actual image data can be output via a call to `pr_dump_image()`. This routine returns 0 unless there is an error, in which case it is PIX_ERR. It cannot write the image in a non-standard (filtered) format, since by the time it is called the raster file header has already been written.

Since these routines sequentially advance the output file's write pointer, `pr_dump_image()` must be called after `pr_dump_header()`.

6.4. Reading Parts of a Raster File

The following routines are available for reading the various parts of a raster file. Many of these routines are used to implement `pr_load()`. Since these routines sequentially advance the input file's read pointer, rather than doing random seeks in the input file, they should be called in the order presented below.

Read Header from Raster File

```
int pr_load_header(input, rh)
FILE *input;
struct rasterfile *rh;
```

The raster file header can be read by calling `pr_load_header()`. This routine reads the header from the specified input, checks it for validity and initializes the specified `rasterfile` structure from the header. The return value is 0 unless there is an error, in which case it is PIX_ERR.

Read Colormap from Raster File

```
int pr_load_colormap(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t *colormap;
```

If the header indicates that there is a non-empty set of colormap values, the values can be read by calling `pr_load_colormap()`. If the specified colormap is NULL, this routine skips over the colormap values by reading and discarding them. If the type and length values in the `colormap` structure do not match the input file, `pr_load_colormap()` allocates space for the colormap with `malloc`, reads in the file's colormap, and modifies `colormap` argument pointer to point to the freshly loaded colormap before returning. If this occurs, the space allocated can be released with a `free(colormap->map[0])`.

The return value is 0 unless there is an error, in which case it is PIX_ERR.

Read Image from Raster File

```
Pixrect *pr_load_image(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t *colormap;
```

An image can be read by calling `pr_load_image()`. If the input is a standard raster file type, this routine reads in the image directly. Otherwise, it writes the header, colormap, and image into the appropriate filter and then reads the output of the filter. In this case, both the raster file and the colormap structures are modified as side-effects of calling this routine. In either case, a pixrect is dynamically allocated to contain the image, the image is read into the pixrect, and the pixrect is returned as the result of calling the routine. If there is an error, the return value is NULL.

Read Standard Raster File

```
Pixrect *pr_load_std_image(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t colormap;
```

If it is known that the image is from a standard raster file type, then it can be read in by calling `pr_load_std_image()`. This routine is identical to `pr_load_image()`, except that it does not invoke a filter on non-standard raster file types.

A

Writing a Pixrect Driver

A

Writing a Pixrect Driver

Sun has defined a common programming interface to pixel-addressable devices that enables device-independent access to all Sun frame buffers. This interface is called the `pixrect` interface. Existing Sun supported software systems access a frame buffer through the pixrect interface. Sun encourages customers with other types of frame buffers (or other types of pixel-addressable devices) to provide a pixrect interface to these devices.

This chapter contains auxiliary material describing how to write a pixrect driver, and is therefore of interest only to pixrect driver implementors. It is assumed that you have already read Chapter 3, *Pixrect Operations* which describes the programming interface to the basic operations that must be provided in order to generate a complete pixrect implementation. It is also assumed that you have a copy of *Writing Device Drivers* The section in that manual on writing the kernel device driver portion of the pixrect implementation is important.

Topics covered in this chapter are as follows:

1. Instructions for installing a new pixrect driver into the software architecture so that it may be used in a device-independent manner.

2. Additional utilities and conventions that may be of use to the pixrect driver implementor.

The actual source code that is presented here is boiler-plate, i.e., almost every pixrect driver implementation will be similar. A complete source example for an existing pixrect driver would probably expedite the development of your own driver. The complete device-specific source files for the Sun-1 color frame buffer pixrect driver are available as a source code purchase option (available without a UNIX source license).

A.1. Prerequisites

These are the tools and pieces that you will need before assembling your pixrect driver:

- The following documents are recommended reading:

  SunView 1 Programmer's Guide

  SunView 1 System Programmer's Guide

  Writing Device Drivers

PROM User's Manual

Writing Device Drivers

SunOS Reference Manual

Debugging Tools

Sun3 Architecture Manual

Sun4 Architecture Manual

- You need to know how to drive the hardware of your pixel-addressable device. At a minimum, a pixel-addressable device must have the ability to read and write single pixel values. (It is possible to have a device that doesn't meet the minimum requirements of a pixel-addressable device. We will not discuss any of the ways that such a device might emulate the minimum requirements).
- You must have a UNIX kernel building environment. No extra source is required.
- You must have the current Pixrect Library file and its accompanying header files. No extra source is required.
- You are a experienced C programmer.
- You are familiar with the C-shell `csh`, and the `ed` editor.
- You are using a Sun-3 family workstation.

If you are using a Sun-4 workstation, substitute `sun4` for references to `sun3` in this chapter. The only exception is the discussion of the `GENERIC` configuration file.

A.2. Overview and Assumptions

A pixrect device driver has three components:

1. The Unix device driver of the device.
2. The device-specific implementation of the pixrect functions.
3. The kernel pixrect, to be explained later in the document.

If you are not comfortable with the `ed` editor, read the `ed` man page. It is a simple and straight-forward line editor, and it is available in single user mode.

This chapter describes a directory hierarchy on which the software development is conducted. The emphasis of this document is on methodology, rather than writing a specific driver or implementing pixrects. These purposes are served by the *Writing Device Drivers* manual and the other chapters in this manual.

Approach Outline

The approach used in this chapter is incremental. Each addition is built on a solid, tested software base. The approach is outlined below:

1. Prepare the directory structure needed build a new kernel. This kernel this then built with no new drivers. The purpose of this step is to make sure the directory structure works.

2. Add one fool-proof device driver to this kernel. The purpose of this step is making sure you know how to add a new device driver to the kernel.
3. Make a normal Pixrect Library, to prepare the pixrect for the new device.
4. Do a dummy implementation of the device pixrect to make sure the system works.
5. Write the real device driver.
6. Finish off the device pixrect.
7. Make a special version of `suntools` based on this new pixrect.

A.3. Preparing the System

You must prepare the system to add the new device driver (since it will go through a lot of modification). The system on which you are writing the driver will be rebooted many times.

It is a good idea to put the driver source code on a server, and then mount it. There is less chance of losing files that way.

To set up the working directory, do the following:

```
example% cd DEVELOPMENT_DIRECTORY
example% mkdir sys; cd sys
example% foreach d (/usr/sys/*)
ln -s $d .
end
example% rm sun sundev sun3
example% tar cf - -C /usr/sys sun sundev sun3 | tar xvf -
example% cd sun3
example% rm -rf OBJ; ln -s /usr/sys/sun3/OBJ .
```

You have created a `sys` directory that makes symbolic links to most of the subdirectories under `/usr/sys`. The exceptions are the `sun`, `sundev` and `sun3` directories. These files are copied into your staging area. Note that the `sun3/OBJ` directory is not copied.

The idea is to duplicate the directory structure of a source machine, but not to copy every file. This saves disk space.

```
example% cd sun3/conf
example% config GENERIC
example% cd ../GENERIC
example% make
```

When the build completes, you should have a new "vmunix" kernel. Try running it:

```
example% /bin/su
Password: Enter root password
example# mv /vmunix{,-}
example# cp {,/}vmunix
example# /etc/fastboot
```

When the system comes up, you will be running the kernel you just built. Everything should run normally. If you see problems, review the steps above and try again.

A.4. A Skeleton Driver

It is now time to name your pixrect device. A pixrect device has several names:

1. A special file name, as mentioned in the /dev directory. The name has no length restriction, but is usually only 4 or 5 letters long.
2. A device driver file name. This is usually the same as the special file name.
3. A pixrect device name. This name is usually two letters followed by a digit.
4. The hardware name. This is the name referred to the hardware board.

As the implementor of pixrect and driver, it is up to you to provide these names. The names do not need to be the same. Exercise good judgment and avoid confusing names. The name of our example device is `bwfb`. The device driver source file is called `bwfb.c`, the device special file is `/dev/bwfb0` and the pixrect device subdirectory is named `libpixrect/bwfb`. Substitute `bwfb` with your own device name as you work through this chapter.

The `bwfb` is a skeleton device driver based on the dumb, monochrome frame buffer. Although your device almost certainly differs, follow the steps in this chapter anyway. We are not trying to write the device driver of your device yet. This chapter only shows how to add a device driver to the kernel.

The files you will modify are:

- `sun/conf.c`
- `/dev/MAKEDEV`
- `sun3/conf/GENERIC`
- `sun3/conf/files`

The file you will create is:

- `sundev/bwfb.c`

You need to inquire of your hardware team to find out the page type, base address, interrupt vector and ID of your device.

The information for our example device, `bwfb` is shown below.

Page Type

The page type of this device is *obmem*. To learn about page types, see the *Sun3 Architecture* manual.

Base Address

The base address of our device is `0xFF000000`. The size of the device is `1152*900/8`. This means the device responds when someone accesses physical addresses within this range.

Interrupt

The device does not interrupt.

Device Id

There is no other way to identify our device. You simply must assume that any hardware that responds to the correct range of addresses is your device.

You can now try to add a dummy driver to the kernel and see if everything still works. You need to modify all the files listed previously except `sundev/bwfb.c`. The file `sundev/bwfb.c` contains all the device driver functions. We will concentrate on this file later.

conf.c

The first file to edit is sun/conf.c. This file has all the device drivers the kernel can possibly use. You need to modify it in two places. You will be adding the bwfb driver functions to it. See the README file in that directory for more information.

You must first declare the driver functions.

```
example% ed conf.c << 'EDEND'
/^struct bdevsw/-1
a
#include "bwfb.h"
#if NBWFB > 0
extern int bwfbopen(), bwfbclose();
extern int bwfbioctl(), bwfbmmap();
#else
#define bwfbopen nodev
#define bwfbclose nodev
#define bwfbioctl nodev
#define bwfbmmap nodev
#endif

w
q
EDEND
```

Next, you add your device to the cdevsw table.

```
example% ed conf.c << 'EDEND'
/nchrdev = sizeof/-2
a
    {
        bwfbopen, bwfbclose, nodev, nodev,  /* MAJ */
        bwfbioctl, nodev, seltrue, bwfbmmap,
        0,
    },
w
q
EDEND
```

This file is really a table of all the device drivers. The functions of each driver are collected into an array called cdevsw (character device switch table). Each element of this array represents one device driver and the sequence number is the *major number* of the device.

It is important to insert a comment regarding the major number of your device for others who may read the code later. Replace the MAJ in the comment with the increment of the previous driver. We will refer to this number simply as MAJ.

MAKEDEV

Knowing the major number, it is time to modify the `/dev/MAKEDEV` shell script, which is used to make the device node.

```
example% ed /dev/MAKEDEV << 'EDEND'
/^local/-1
a
bwfb*)
    mknod bwfb0 c major_number 0
    chmod 666 bwfb0
    ;;

w
q
EDEND
```

After editing the MAKEDEV script, you can make the device.

```
example% cd /dev; MAKEDEV bwfb0
```

files

The `/usr/sys/sunX/conf/files` file lists all the source files necessary to make the kernel. Add the new driver source file to it.

```
example% ed sun3/conf/files << 'EDEND'
$-1a
sundev/bwfb.c   optional bwfb device-driver
w
q
EDEND
```

GENERIC

The last editing step is to add the device into the configuration file `GENERIC`, or to any other standard configuration files, as appropriate. Add your device next to the `bwtwo0`.

```
example% ed sun3/conf/GENERIC << 'EDEND'
/bwtwo0/
a
device      bwfb0 at obmem 1 csr 0xff000000
w
q
EDEND
```

sundev/bwfb.c

Now you are ready to rebuild the kernel, but you still do not have an actual driver. You can start with the following template:

```
example% cat > sundev/bwfb.c
#include "bwfb.h"

#include <sys/param.h>
#include <sys/buf.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <sys/map.h>
#include <sys/vmmac.h>

#include <machine/pte.h>
#include <machine/mmu.h>
#include <machine/cpu.h>
#include <sundev/mbvar.h>

#include <machine/eeprom.h>
#include <machine/enable.h>

#define BWFB_PROBESIZE NBPG
/*
 * Driver information for auto-configuration stuff.
 */
int bwfbprobe(), bwfbattach();
struct  mb_device *bwfbinfo[NBWFB];
struct  mb_driver bwfbdriver = {
    bwfbprobe, 0, bwfbattach, 0, 0, 0,
    BWFB_PROBESIZE, "bwfb", bwfbinfo, 0, 0, 0,
};

bwfbopen(dev, flag)
    dev_t dev;
    int flag;
{
    return 0;
}

/*ARGSUSED*/
```

```
bwfbclose(dev, flag)
    dev_t dev;
    int flag;
{
    return 0;
}

/*ARGSUSED*/
bwfbmmap(dev, off, prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return -1;
}

/*
 * Determine if a bwfb exists at the given address.
 * If it exists, determine its size.
 */
/*ARGSUSED*/
bwfbprobe(reg, unit)
    caddr_t reg;
    int unit;
{

    /* What should this return - a non-zero if device */
    return BWFB_PROBESIZE;
}

bwfbattach(md)
    register struct mb_device *md;
{
    return 0;
}

/*ARGSUSED*/
bwfbioctl(dev, cmd, data, flag)
    dev_t dev;
    int cmd;
    caddr_t data;
    int flag;
{
    return ENOTTY;
}
^D
```

Now you are ready to generate another kernel.

```
example% cd sun3/conf
example% config GENERIC
example% cd ../GENERIC
example% make
example% /bin/su
Password: Enter password
example# cp vmunix /
example# /etc/fastboot
```

During the build, you should fix any syntax errors in the `bwfb.c` driver file. After rebooting, the login message should look like the following:

```
SunOS Release 4.0 (GENERIC) #2: Current time and date
```

The important part is the `(GENERIC) #2` which means that this is the second time you are making the `GENERIC` kernel. (The first time was the test run.)

Recommended Reading

At this point, you should read the `adb(1)` manual and the appropriate man pages, along with the `kadb(1)` man pages.

As a test, compile and run the following program.

```
main()
{
    close(open("/dev/bwfb0", 2));
}
```

Use `kadb` to make sure that the device driver `bwfbopen()` function is called.

A.5. A Skeleton Pixrect Device Module

The goal of this section is to get the following program working:

```
main ()
{
    Pixrect *pr;

    if ((pr = pr_open ("/dev/bwfb0")) == NULL)
       return 1;
    pr_close (pr);
    return 0;
}
```

You need to establish a bare skeleton of the development structure to which real device-dependent code can be added later. After this structure has been erected, you simply keep modifying and enhancing it until you have a satisfactory driver. Up to this point, you do not need to make any design decisions.

pr_open

The function `pr_open` is called by application programs to create a pixrect in a device-independent manner. When the function is called, a device name is passed to it (`/dev/bwfb0`, in our case). It opens the device with system call `open`, and receives a file descriptor in return. If the descriptor is valid (greater than or equal to zero), it uses the descriptor to make several `ioctl` system calls. The purpose of these calls is to:

1. Make sure the device is a valid frame buffer.
2. Identify the device.
3. Find the configurable parameters of the device in the kernel.

The device is identified by a frame buffer type, which is a small integer defined in the file `<sun/fbio.h>`. A new number must be created before a new device-dependent module is implemented. This number is obtained from the kernel via the `ioctl` calls.

The device type number is then used as an index for an internal table of device-dependent functions. The function is called to create and initialize the pixrect. If successful, it is returned to the caller of `pr_open`. The device-dependent function is usually named *devname*`_make` where *devname* is `bwfb` in our case.

When the device-dependent function is called, it first maps in the frame buffer and its control registers. Then it allocates the device-dependent data structure and properly initializes it. Finally, it initializes the `pixrectops` vector, which contains the device-dependent functions for standard pixrect operations.

Pixrect Staging Area

Like the staging area of the device driver, you now need to create a similar structure for the Pixrect Library. In our examples, the directory `DEVELOPMENT_DIRECTORY` is exactly the same directory used before. The relative positions of the subdirectories are all that is important.

The following commands extract the object files from the SunOS release's Pixrect Library and put them in the directory `OBJ`. The file `__.SYMDEF` is created by the ranlib command. The file pr_makefun.o has be to rebuilt to add our new device. The directory pixrect is the directory for all the include files.

```
example% cd DEVELOPMENT_DIRECTORY
example% mkdir usr.lib ; cd usr.lib
example% mkdir libpixrect; cd libpixrect
example% mkdir OBJ pixrect bwfb
example% cd OBJ
example% ar x /usr/lib/libpixrect.a
example% rm __.SYMDEF pr_makefun.o
example% cd ../pixrect
example% foreach f (/usr/include/pixrect/*)
    ln -s $f .
end
```

You can now create the skeleton file pr_makefun.c and prepare the Makefile for the library:

```
example% cd bwfb
example% cat > pr_makefun.c
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <sun/fbio.h>

Pixrect *bw2_make();
Pixrect *cg2_make();
Pixrect *cg3_make();
Pixrect *cg4_make();
Pixrect *cg6_make();
Pixrect *cg8_make();
Pixrect *cg9_make();
Pixrect *gp1_make();
Pixrect *tv1_make();


Pixrect *(*pr_makefun[FBTYPE_LASTPLUSONE])() = {
    0,          /* FBTYPE_SUN1BW */
    0,          /* FBTYPE_SUN1COLOR */
    bw2_make,       /* FBTYPE_SUN2BW */
    cg2_make,       /* FBTYPE_SUN2COLOR */
    gp1_make,       /* FBTYPE_SUN2GP */
    0,          /* FBTYPE_SUN5COLOR */
    cg3_make,       /* FBTYPE_SUN3COLOR */
    cg8_make,       /* FBTYPE_MEMCOLOR == 7*/
    cg4_make,       /* FBTYPE_SUN4COLOR == 8 */
    0,          /* FBTYPE_NOTSUN1 == 9 */
    0,          /* FBTYPE_NOTSUN2 == 10 */
    0,          /* FBTYPE_NOTSUN3 == 11 */
    cg6_make,       /* FBTYPE_SUNFAST_COLOR == 12 */
    cg9_make,       /* FBTYPE_SUNROP_COLOR == 13 */
    tv1_make,       /* FBTYPE_SUNFB_VIDEO == 14 */
    0,          /* 15 */
    0,          /* 16 */
    0,          /* 17 */
    0,          /* 18 */
    0,          /* 19 */
};
^D
```

NOTE *The file* `<sun/fbio.h>` *is included in this file. This will be discussed later.*

The `Makefile` is shown below. The macro `BWFBSRC` is the hook used to add our device module.

```
example% cat > Makefile
BWFBSRCS =
BWFBOBJS = $(BWFBSRCS:.c=.o)

CPPFLAGS =-I../../sys -I..
CFLAGS   =-g

libpixrect.a: pr_makefun.o $(BWFBOBJS)
	ar ru $@ pr_makefun.o $(BWFBOBJS) ../OBJ/*.o
	ranlib $@

^D
example% make
```

The build should create a `libpixrect.a` file, which is a new version of the Pixrect Library, but with nothing new added to it. If you link a pixrect application to it, it should run as if it were linked to the regular library.

```
example% cc -o foo foo.c libpixrect.a
```

or

```
example% cc -o foo foo.c -LDEVELOPMENT_DIRECTORY/usr.lib/libpixrect/bwfb -lpixrect
```

Only after the program `foo` runs without error should you go to the next step.

../../sys/sun/fbio.h

The file `fbio.h` should reside in the `../../sys/sun` directory. The first version of the file can be copied from `/usr/include/sun/fbio.h`. The last few lines of the file should look like this:

```
/* frame buffer type codes */
#define FBTYPE_SUN1BW        0   /* Multibus mono */
#define FBTYPE_SUN1COLOR     1   /* Multibus color */
#define FBTYPE_SUN2BW        2   /* memory mono */
#define FBTYPE_SUN2COLOR     3   /* color w/rasterop chips */
#define FBTYPE_SUN2GP        4   /* GP1/GP2 */
#define FBTYPE_SUN5COLOR     5   /* RoadRunner accelerator */
#define FBTYPE_SUN3COLOR     6   /* memory color */
#define FBTYPE_MEMCOLOR      7   /* memory 24-bit */
#define FBTYPE_SUN4COLOR     8   /* memory color w/overlay */

#define FBTYPE_NOTSUN1           9       /* reserved for customer */
#define FBTYPE_NOTSUN2           10      /* reserved for customer */
#define FBTYPE_NOTSUN3           11      /* reserved for customer */

#define FBTYPE_SUNFAST_COLOR     12      /* accelerated 8bit */
#define FBTYPE_SUNROP_COLOR      13      /* MEMCOLOR with rop h/w */
#define FBTYPE_SUNFB_VIDEO       14      /* Simple video mixing */
#define FBTYPE_SUNGIFB           15      /* medical image */
#define FBTYPE_SUNGPLAS          16      /* plasma panel */
```

```
#define FBTYPE_RESERVED3         17      /* reserved, do not use */
#define FBTYPE_RESERVED2         18      /* reserved, do not use */
#define FBTYPE_RESERVED1         19      /* reserved, do not use */

#define FBTYPE_LASTPLUSONE       20      /* max number of fbs (change as add) */
```

Define a new constant, say `FBTYPE_BWFB`, for `bwfb`, incrementing the value of `FBTYPE_LASTPLUSONE`, if necessary. Assume that `FBTYPE_RESERVED3` is used by your device.

```
#define FBTYPE_BWFB     17
```

pr/pr_makefun.c

The next step is to add the function `bwfb_make` (which you have not written yet) to the file `pr/pr_makefun.c`. You need to make sure the seventeenth entry of the `pr_makefun` table is `bwfb_make`.

```
example% ed pr_makefun.c << 'EDEND'
/pr_makefun[FBTYPE_LASTPLUSONE]/-3a
Pixrect *bwfb_make();
/17/
d
i
    bwfb_make,       /* FBTYPE_BWFB == 17 */
w
q
EDEND
```

The function `bwfb_make()` should reside in the file `bwfb.c`. For now, it can be a empty place holder function.

```
example% cat > bwfb.c
#include <stdio.h>
#include <pixrect/pixrect.h>

Pixrect *
bwfb_make()
{
    return NULL;
}
^D
```

You can now change your `Makefile` to include `bwfb.c`.

```
example% ed Makefile < 'EDEND'
/BWFBSRCS/s/$/bwfb.c/
w
q
EDEND
```

You can now rebuild the library. Our example program (shown previously) should run.

```
example% make
cc -g -I../../sys -I.. -sun4 -c  pr_makefun.c
cc -g -I../../sys -I.. -sun4 -c  bwfb.c
ar ru libpixrect.a pr_makefun.o bwfb.o ../OBJ/*.o
ranlib libpixrect.a
```

Recommended Reading

At this point, you should study the contents of the following files: `sys/sun/fbio.h`, `libpixrect/pr/pr_open.c`, `libpixrect/pr/pr_make.c`, and `libpixrect/bw2/bw2.c`.

You should also learn the syntax and semantics of the system calls `mmap(2)` and `ioctl(2)`. These two functions are vital to the pixrect driver. Become familiar with the mechanism of a system call. You need to know what means to say "it is in the kernel". You can learn about this in section 2 of the man pages.

A.6. Adding Flesh to the Skeleton

This section shows how to add the device-dependent modules to the pixrect driver. It requires both kernel and user programming. The developer should expect to frequently reboot the system.

The job is much easier if you set up the proper development environment beforehand, and if you are familiar with the tools. At this point, you should have mastered `dbx`, `adb`, and `kadb`. It is quite difficult to do kernel or library debugging. Code should be written with ease-of-debugging in mind; you can maximize the driver's performance later.

bwfb_ops.c

This file defines the *op vector*, which is the collection of device-dependent functions for basic pixrect operations. Since our device is a simple one, it can be derived from the generic memory pixrect software. The first version of the file is:

```
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/bwfbvar.h>
#include <pixrect/memvar.h>

struct pixrectops bwfb_ops = {
    mem_rop,
    mem_stencil,
    mem_batchrop,
    0,
    bwfb_destroy,
    mem_get,
    mem_put,
    mem_vector,
    mem_region,
    mem_putcolormap,
    mem_getcolormap,
    mem_putattributes,
    mem_getattributes
};
```

This file includes `<pixrect/bwfb.h>`, which should contain:

```
#ifndef bwfbvar_DEFINED
#define bwfbvar_DEFINED

extern  struct pixrectops bwfb_ops;

#ifndef KERNEL
struct  pixrect *bwfb_make();
int bwfb_destroy();
#endif

#endif  bwfbvar_DEFINED
```

NOTE *The include file may be used during kernel building. It is necessary to isolate it from user declarations with* `#ifdef`*'s. In a real driver, this file usually defines device-dependent data structures and constants.*

The last step is to fill in the `bwfb.c` file with the contents of the `bwfb_amke()` and `bwfb_destroy()` functions.

bwfb_make/bwfb_destroy

The following steps entail overwriting the previous version of `bwfb.c`; it was only used as a place holder.

```
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_impl_make.h>
#include <pixrect/memvar.h>
#include <pixrect/bwfbvar.h>

static  struct pr_devdata *bwfbdevdata;

Pixrect *
bwfb_make(fd, size, depth)
    int fd;
    struct  pr_size size;
    int depth;
{
    register int w = size.x, h = size.y;
    register Pixrect *pr;
    struct  pr_devdata *dd;
    register int linebytes;

    /*
     * Allocate/initialize pixrect and get virtual address for it.
     */
    linebytes = mpr_linebytes(w, depth);
    if ((pr = pr_makefromfd(fd, size, depth, &bwfbdevdata, &dd,
        h * linebytes, sizeof(struct mpr_data), 0)) != 0) {

        register struct mpr_data *md;

        pr->pr_ops = &bwfb_ops;

        md = (struct mpr_data *) pr->pr_data;
        md->md_linebytes = linebytes;
        md->md_image = (short *) dd->va;
        md->md_offset.x = 0;
        md->md_offset.y = 0;
        md->md_primary = -1 - dd->fd;
        md->md_flags = MP_DISPLAY;  /* pr is display dev */
    }
    return pr;
}


bwfb_destroy(pr)
```

```
    Pixrect *pr;
{
    if (pr != 0) {
        register struct mpr_data *md;

        if ((md = mpr_d(pr)) != 0) {
            if (md->md_primary)
                (void) pr_unmakefromfd(-1 - md->md_primary,
                    &bwfbdevdata);
            free((char *) md);
        }
        free((char *) pr);
    }
    return 0;
}
```

Now edit the Makefile, and create symbolic links to the include file.

```
example% ln -s ../bwfb/bwfbvar.h ../pixrect
example% ed Makefile << 'EDEND'
/BWFBSRC/s/$/ bwfb_ops.c/
w
q
EDEND
example% make
cc -g -I../../sys -I.. -sun4 -c  bwfb.c
cc -g -I../../sys -I.. -sun4 -c  bwfb_ops.c
ar ru libpixrect.a pr_makefun.o bwfb.o bwfb_ops.o ../OBJ/*.o
ranlib libpixrect.a
example%
```

Back to the driver

The last step is to add enough functionality to the driver so that the `pr_open` sequence will work. You need to add the support for the FBIOGATTR, command of the `ioctl` system call, and the mmap system call. The modification of the `bwfbioctl` and `bwfbmmap` device driver functions is shown below:

```
#define BWFB_SIZE (1152 * 900 / 8)

/*ARGSUSED*/
bwfbioctl (dev, cmd, data, flag)
    dev_t           dev;
    int             cmd;
    caddr_t         data;
    int             flag;
{
    switch (cmd) {
    case FBIOSATTR:
    break;
    case FBIOGATTR:{
        register struct fbgattr *gattr =
```

```
                    (struct fbgattr *) data;

        gattr->owner = 0;
        gattr->real_type = FBTYPE_BWFB;
        gattr->fbtype.fb_type = FBTYPE_BWFB;
        /* change the followings for the real device */
        gattr->fbtype.fb_height = 900;
        gattr->fbtype.fb_width = 1152;
        gattr->fbtype.fb_depth = 1;
        gattr->fbtype.fb_cmsize = 2;
        gattr->fbtype.fb_size = BWFB_SIZE;
        gattr->sattr.flags = 0;
        gattr->sattr.emu_type = -1;
        gattr->sattr.dev_specific[0] = 0;
        gattr->emu_types[0] = -1;
        break;
    }
    default:
    return ENOTTY;
    }
    return 0;
}

/*ARGSUSED*/
bwfbmmap (dev, off, flag)
    dev_t           dev;
    off_t           off;
    int             flag;
{
    /* re-write for the real device */
    if (off < 0 || off >= BWFB_SIZE)
    return -1;
    return PGT_OBMEM | btop (off + 0xff000000);
}
```

After the driver is modified, rebuild the kernel and reboot the system.

```
example% cd sys/sun3/GENERIC
example% make
example% cp vmunix /
example% /etc/reboot
```

Now, enter the program below, a modified version of the test program shown at the start of the *A Skeleton Pixrect Device Module* Section, into the top directory.

```
example% cat > pixrect_test.c
#include <stdio.h>
#include <pixrect/pixrect_hs.h>
main ()
{
    Pixrect        *pr;

    if ((pr = pr_open ("/dev/bwfb0")) == NULL)
    printf ("Failed\n");
    else {
    printf ("Made it\n");
    pr_close (pr);
    }
    return 0;
}
^D
example% cc -o pixrect pixrect_test.c usr.lib/libpixrect/bwfb/lib
pixrect.a
```

If the program prints "`Made it`", you have successfully completed the basic driver.

A.7. The Real Driver

Now that the dummy driver and pixrect are done, proceed to the real driver. At this time, you should; be very comfortable with rebuilding the kernel, using `kadb` to set break points in kernel routines and examine variables, using `printf()` statements in the driver code to discover where things are going wrong.

Before you start writing any software, you should understand the device thoroughly. Become as familiar as possible with the hardware manual for the device. You can develop some of the code for the functions described in the previous section of this chapter, before you ever see the hardware. This would depend upon your particular skill, however, and it may be more fruitful to proceed with hardware in hand.

Visual Inspection of the Hardware

When the board arrives, keep in mind that the hardware may not be bug-free or completely and correctly documented. If there are problems, investigate the possibility of hardware bugs first.

Inspect the hardware closely. Look for loose parts or broken wires (press in socketed IC's). Find out if the backplane configuration must be changed. Find out if jumpers or dip switches need to be set. Identify the jumpers used for address changes, interrupt vectors, or otherwise relevant to software development. If everything seems fine, halt the system, power it down, and plug in the board, making sure it is properly seated.

PROM Monitor

The PROM monitor is a powerful tool for driver development. It is the software closest to the hardware device. It is also the most reliable and convincing tool for determining if the hardware is functioning according to the spec.

Power up the system, then halt it when the self test starts. You should see the PROM Monitor prompt, ">".

The PROM commands you should be familiar with are:

- Address mapping commands
- Data reading and writing commands
- Self-testing commands.

First, map the physical addresses of the hardware device to the virtual addresses of your choice. When this mapping is done, you can read from and write to the device using virtual addresses. If you find something wrong, determine if defects exist in the system or the hardware device.

NOTE *These commands can vary from architecture to architecture. This document uses Sun3 PROM commands as examples.*

PROM Monitor Command Summary		
Command	**Syntax**	**Notes**
m	m address	display/modify segment map
p	m address	display/modify page table
s	m mode	display/modify address space
^t	^t address	display address mapping
o	e address [value]	display/modify 1 bytes
l	e address [value]	display/modify 2 bytes
l	l address [value]	display/modify 4 bytes
f	f start end pattern size	block write pattern
v	f start end size	block display
x	x	extended test

Monitor Command Example

Imagine there is a VME 32/32 device at the physical address 0x8000000. At this location, there is a device ID register whose value is supposed to be 0x2. This example shows how to use the PROM monitor to determine if this is indeed the case.

First, you must enter the PROM monitor, then put the system into the supervisor space with the s command. Change the mapping of the virtual address `0xE000000` to physical address `0x8000000`, then read 4 bytes from that address:

```
example% su
Password: enter superuser (root) password
example# halt
system shuts down...
> s 5              system in supervisor state (s B for Sun4)
> pE000000 fc004000        change the mapping
> lE000000          read 4 bytes
```

The `p` command establishes the mapping of virtual addresses to physical ones. The physical address, however, should be converted into *PTE* format. The conversion can be done as follows:

```
PHYSICAL=0x8000000
TYPE=VME32D32
if TYPE == OBMEM
    PTE_MASK=0XF0000000
else if TYPE == OBIO
    PTE_MASK=0XF4000000
else if TYPE == VMEXXD16
    PTE_MASK=0XF8000000
else if TYPE == VMEXXD32
    PTE_MASK=0XFC000000

if TYPE == VME24DXX
    PHYSICAL += 0XFF000000
else if TYPE == VME16DXX
    PHYSICAL += 0XFFFF0000

PHYSICAL >>= 13
PTE = PHYSICAL | PTE_MASK
```

The virtual address `0xE000000` now points to the top of the page to which the device has been mapped. The virtual address of the device is now at this physical location:

```
0xE000000 + (PHYSICAL & 0x1FFF)
```

where `PHYSICAL` is now `0x8000000`.

bwfbreg.h

This include file resides in the `sundev` directory and should be installed in `/usr/include/sundev` by the `Makefile` in the `/usr/src/sys/sundev` directory. This file has two purposes:

- To define the hardware device in a software structure.
- To provide an abstract model of the hardware for the user's program.

You should read through the hardware specification and define a structure for each logical unit of the device. A constant should be defined for each state, magic number (like the ID register) and physical address. Finally, a structure should be defined for the entire device. The naming of the constants and structures should be unique and descriptive. The style should follow local convention. The structures should define both the accessing restrictions and logical meaning of the registers.

As an example, imagine the device has a register 32 bits wide. Bits 0 to 8 act as the device's identification number. Bit 31 is the reset bit. The rest are have no effect when written to, and always read out as 1's. A good structure would be:

```
union dev_reg {
    struct {
        int reset:1;
        :23;
        int id:8;
    } id_reset;
    int access_packed;
};
```

In this way, the register can be read using *access_packed*, and examined bit by bit with the structure *id_reset*.

bwfbprobe

This probe function should determine if the device really exists in the system. If it does, the function should return a non-zero value. This function should read the ID register of the device, if there is any, and as many other device registers as necessary in order to determine that it is indeed the expected device.

bwfbattach

The `bwfbattach` function is called if `bwfbprobe` returns non-zero. It can be developed in user space (so that it can be debugged with dbx), then moved to the kernel. Write a user program that maps the device into the address space and initializes the device. This program will be the skeleton of the `bwfbattach()` function and will be very useful. Implement some diagnostic functionality into this program. That way, if you suspect the device is not working correctly, you can run this program and check immediately.

bwfbmmap

The `bwfbmmap` function maps the relevant device memory and registers into user address space. In order to write this function correctly, you must answer some questions about the device hardware. How do different portions of the device get accessed? Can they be memory mapped? What are the address offsets from the base address?

Features and Trap Doors

To write an efficient pixrect driver, it is important to take advantage of any hardware features, and avoid obstacles. Ask yourself: How can various pixrect operations be done? Is there any hardware assistance for them? Are there any hardware obstacles if they must be done in software?

Once you understand the address mapping, you can write the `bwfbreg.h` file. This file describes the device in software. It is usually a replication of the address mapping part of the hardware spec. Once this file is written, the `bwfbmmap` function can be completed. After writing the `mmap(2)` function, you should try to do the following:

Write a user program that maps in the device and accesses all its parts. This program should be interactive. It should accept commands from the user and interact with the device as requested. It should be a window-based program.

The `mmap` function is an important one. It provides you with access to the device from user space. With the interactive program you have written, you can discover the best command sequence to initialize the device, and the best way to identify it. You should now finish off the bwfbprobe and bwfbattach functions. Put several printf statements in the probe and the attach functions. At boot time, you can then make sure these functions are called, and you can ensure the proper behavior.

A.8. Creating the Real Pixrect

When the probe, attach, ioctl, and mmap functions are working, you can start writing the real pixrect software for the device. Most of the software resides in the libpixrect/bwfb directory. Anything outside this directory is discussed in the previous sections of this chapter.

The recommended procedure is:

1. Design and code bwfbvar.h. The most important part of this file is the device private data structure, the structure that pr_data points to.
2. Design and code bwfb_make(), bwfb_region(), and bwfb_destroy(). The destroy function has been partly implemented in the last section.
3. Design and code bwfb_putcolormap(), bwfb_putattributes(), and bwfb_rop(). Edit bwfb_ops.c to add these functions.
4. Write the kernel pixrect by going back to the driver and writing the FBOIGPIXRECT command of the ioctl(2) function. Put pixrect/../bwfb/bwfb_colormap.c and pixrect/../bwfb/bwfb_rop.c into the sun3/conf/files file. The kernel pixrect uses the three functions listed in the previous step.
5. Build a special version of suntools that uses the Pixrect Library you just created. Make certain it runs. it work.
6. Finish off the rest of the pixrect functions. Edit bwfb_ops.c accordingly.

A.9. Implementation Strategy

This is one possible step-by-step approach to implementing a pixrect driver:

- Write and debug pixrect creation and destruction. This involves the pixrect kernel device driver that lets you open(2) and mmap(2) the physical device from a user process. The private bwfb_make routine must be written. The bwfb_region and bwfb_destroy pixrect operation must be written.
- Write and debug the basic pixel rectangular region function. The bwfb_putattributes and bwfb_putcolormap pixrect operations must be written in addition to the bwfb_rop routine.
- Write and debug batchrop routines. The bwfb_batchrop pixrect operation must be written.
- Write and debug vector drawer. The bwfb_vector pixrect operation must be written.
- Write and debug remaining pixrect operations: bwfb_stencil, bwfb_get, bwfb_put, bwfb_getattributes and bwfb_getcolormap.

- If the device is to run with SunView, build a kernel with minimal basic pixel rectangle function for use by the cursor tracking mechanism in the SunView kernel device driver. Also include the colormap access routines for use by the colormap segmentation mechanism in the SunView kernel device driver.
- Load and test SunView programs with the new pixrect driver. Experience has shown that when you are able to run released SunView programs, your pixrect driver is in good shape.

A.10. Files Generated

Here is the list of source files generated that implement the example pixrect driver:

- `bwfbreg.h` - A header file describing the structure of the raster device. It contains macros used to address the raw device.
- `bwfbvar.h` — A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.
- `/sys/sundev/cgone.c` — The pixrect kernel device driver code.
- `bwfb.c` — The pixrect creation and destruction routines.
- `bwfb_region.c` — The region creation routine.
- `pr_makefun.c` — Replaces an existing module and contains the vector of pixrect make operations.
- `bwfb_batch.c` — The batchrop routine.
- `bwfb_colormap.c` — The colormap access and attribute setting routines.
- `bwfb_getput.c` — The single pixel access routines.
- `bwfb_rop.c` — The basic pixel rectangle manipulation routine.
- `bwfb_stencil.c` — The stencil routine.
- `bwfb_vec.c` — The vector rendering routine.
- `bwfb_curve.c` — The curved shape routine.
- `bwfb_polyline.c` — The polyline routine.

A.11. Access Utilities

This section describes utilities used by pixrect drivers. The pixrect header files `memvar.h`, `pixrect.h` and `pr_util.h` contain useful macros that you should familiarize yourself with; they are not documented here. Look for the files in `/usr/include/pixrect`.

```
pr_clip(dstp, srcp)
     struct  pr_subregion *dstp;
     struct  pr_prpos *srcp;
```

`pr_clip` adjusts the position and size of `dstp`, the destination pixrect subregion, to fall within `dstp->pr`. If `*scrp`, the source pixrect position, is not zero then the position of the source is clipped to fall within `dstp`.

Two operations on reverse video control, `pr_reversesrc()` and `pr_reversedst()`, are provided for adjusting the raster operation code to take into account reverse video monochrome pixrects in either the source or destination pixrect.

```
char    pr_reversedst[16];
char    pr_reversesrc[16];
```

These are implemented by table look-up in which the index into the tables is `(op>>1)&0xF` where `op` is the operation passed into pixrect public procedures. This process can be iterated, e.g.,
`pr_reversedst[pr_reversesrc[op]]`.

A.12. Rop

These are the major cases to be considered with the `pwo_rop()` operation:

- Case 1 -- we are the source for the pixel rectangle operation, but not the destination. This is a pixel rectangle operation from the frame buffer to another kind of pixrect. If the destination is not a memory pixrect, then one will be allocated temporarily. The source will be roped to this temporary pixrect, then back to the destination pixrect on the frame buffer.
- Case 2 -- writing to your frame buffer. This consists of 4 different cases depending on where the data is coming from: from a constant pixel value, from memory, from some other pixrect, and from the frame buffer itself. When the source is some other pixrect, other than memory, ask the other pixrect to read itself into temporary memory to make the problem easier.

A.13. Batchrop

A simple batchrop implementation could iterate on the batch items and call `rop` for each. Even in a more sophisticated implementation, while iterating on the batch items, you might also choose to bail out by calling `rop` when the source is skewed, or if clipping causes you to cut off the negative x axis.

A.14. Vector

There are some notable special cases that you should consider when drawing vectors:

- Handle length 1 or 2 vectors by just drawing endpoints.
- If vector is horizontal, use fast algorithm.
- If vector is vertical, use fast algorithm.

Importance of Proper Clipping

The hard part in vector drawing is clipping, which is done against the rectangle of the destination quickly and with proper interpolation so that the jaggies in the vectors are independent of clipping.

A.15. Colormap

Each color raster device has its own way of setting and getting the colormap.

Monochrome

The convention for monochrome raster devices when `pr_putcolormap()` is called is that if `red[0]` is zero then the display is light on dark; otherwise it is dark on light. The convention for monochrome raster devices when `pr_getcolormap()` is called is that if the display is light on dark then zero is stored in `red[0]`, `green[0]` and `blue[0]`, and —1 is stored in other positions in the colormap. Otherwise, if the display is dark on light, then zero and —1 are reversed.

A.16. Attributes

`pr_getattributes()` and `pr_putattributes()` operations get or set a bitplane mask in color pixrects, respectively.

Monochrome

Monochrome devices ignore `pr_putattribute()` calls that are setting the bitplane mask. Monochrome devices always return 1 when `pr_getattribute()` is asking for the bitplane mask.

A.17. Pixel

`pwo_get()` and `pwo_put()` operations get or set a single pixel, respectively.

A.18. Stencil

In its most efficient implementation, stencil code parallels `rop` code, all the while considering the two-dimensional stencil. One way to implement stencil is to use `rops`. You pay a small efficiency penalty for this. You may not consider it worthwhile to write the special purpose code for the bitmap stencils since they probably will not get used nearly as much as `rop`. Here is the basic idea (`Temp` is a temporary memory pixrect):

```
Temp = Dest
Temp = Dest op Source
Temp = Temp & Stencil
Dest = Dest & ~Stencil
Dest = Dest | Temp
```

i.e.,

```
Dest = (Dest & ~Stencil) | ((Dest op Source) & Stencil)
```

B

Pixrect Functions and Macros

B

Pixrect Functions and Macros

B.1. Making Pixrects

Table B-1 *Pixrects*

Name	*Function*
Create Pixrect	`Pixrect *pr_open(devicename)` `char *devicename;`

Create Secondary Pixrect	`#define Pixrect *pr_region(pr, x, y, w, h)`
	`Pixrect *pr;`
	`int x, y, w, h;`

Release Pixrect Resources	`#define pr_close(pr)` `Pixrect *pr;`
Release Pixrect Resources	`#define pr_destroy(pr)` `Pixrect *pr;`
Subregion Create Secondary Pixrect	`#define Pixrect *prs_region(subreg)` `struct pr_subregion subreg;`
Subregion Release Pixrect Resources	`#define prs_destroy(pr)` `Pixrect *pr;`
Convert 680X0 pixrect to 386i pixrect	`void pr_flip(pr)` `Pixrect *pr;`

B.2. Text

Table B-2 *Text*

Name	Function

Compute Bounding Box of Text String	`pf_textbound(bound, len, font, text)`
	`struct pr_subregion *bound;`
	`int len;`
	`Pixfont *font;`
	`char *text;`

Compute Location of Characters in Text String	`struct pr_size pf_textbatch(where, lengthp, font, text)`
	`struct pr_pos where[];`
	`int *lengthp;`
	`Pixfont *font;`
	`char *text;`

Compute Width and Height of Text String	`struct pr_size pf_textwidth(len, font, text)`
	`int len;`
	`Pixfont *font;`
	`char *text;`

Load Font	`Pixfont *pf_open(name)` `char *name;`
Load Private Copy of Font	`Pixfont *pf_open_private(name)` `char *name;`
Load System Default Font	`Pixfont *pf_default()`
Release Pixfont Resources	`pf_close(pf)` `Pixfont *pf;`

Unstructured Text	`pr_text(pr, x, y, op, font, text)`
	`Pixrect *pr;`
	`int x, y, op;`
	`Pixfont *font;`
	`char *text;`
	
	`pr_ttext(pr, x, y, op, font, text)`
	`Pixrect *pr;`
	`int x, y, op;`
	`Pixfont *font;`
	`char *text;`

Write Text and Background	`pf_text(where, op, font, text)`
	`struct pr_prpos where;`
	`int op;`
	`Pixfont *font;`
	`char *text;`

Table B-2 *Text— Continued*

Name	*Function*

Write Text	`pf_ttext(where, op, font, text)`
	`struct pr_prpos where;`
	`int op;`
	`Pixfont *font;`
	`char *text;`

B.3. Raster Files

Table B-3 *Raster Files*

Name	*Function*

Initialize Raster File Header	`Pixrect *pr_dump_init(input_pr, rh, colormap, type, copy_flag)`
	`Pixrect *input_pr;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`
	`int type, copy_flag;`

Read Colormap from Raster File	`int pr_load_colormap(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Read Header from Raster File	`int pr_load_header(input, rh)`
	`FILE *input;`
	`struct rasterfile *rh;`

Read Image from Raster File	`Pixrect *pr_load_image(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Read Raster File	`Pixrect *pr_load(input, colormap)`
	`FILE *input;`
	`colormap_t *colormap;`

Read Standard Raster File	`Pixrect *pr_load_std_image(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t colormap;`

Write Header to Raster File	`int pr_dump_header(output, rh, colormap)`
	`FILE *output;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Write Image Data to Raster File	`int pr_dump_image(pr, output, rh)`
	`Pixrect *pr;`
	`FILE *output;`
	`struct rasterfile *rh;`

Write Raster File	`int pr_dump(input_pr, output, colormap, type, copy_flag)`
	`Pixrect *input_pr;`
	`FILE *output;`
	`colormap_t *colormap;`
	`int type, copy_flag;`

B.4. Memory Pixrects

Table B-4 *Memory Pixrects*

Name	*Function*

Create Memory Pixrect from an Image	`Pixrect *mem_point(width, height, depth, data)`
	`int width, height, depth;`
	`short *data;`

Create Memory Pixrect	`Pixrect *mem_create(w, h, depth)` `int w, h, depth;`

Create Static Memory Pixrect	`#define mpr_static(name, w, h, depth, image)`
	`int w, h, depth;`
	`short *image;`

Get Memory Pixrect Data Bytes per Line	`#define mpr_linebytes(width, depth)` `( ((pr_product(width, depth)+15)>>3) &~1)`
Get Pointer to Memory Pixrect Data	`#define mpr_d(pr)` `((struct mpr_data *)(pr)->pr_data)`

Variations for the Sun386i:

- `mem_point()` on the Sun386i does not flip the bitmap pointed to by `*data`. The pixrect structure returned does not have the `MP_STATIC` or the `MP_I386` flag set.
- `mem_create()` on the Sun386i creates an empty pixrect with the `MP_I386` flag set.
- `mpr_static()` on the Sun386i creates a pixrect with both the `MP_I386` and `MP_STATIC` flags set.

B.5. Colormaps and Bitplanes

Table B-5 *Colormaps and Bitplanes*

Name	*Function*

Exchange Foreground and Background Colors	`pr_reversevideo(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Get Colormap Entries	`#define pr_getcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Get Plane Mask	`#define pr_getattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Set Background and Foreground Colors	`pr_blackonwhite(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Set Colormap Entries	`#define pr_putcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Set Foreground and Background Colors	`pr_whiteonblack(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Set Plane Mask	`#define pr_putattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Subregion Get Colormap Entries	`#define prs_getcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Subregion Get Plane Mask	`#define prs_getattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Subregion Set Colormap Entries	`#define prs_putcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Table B-5 *Colormaps and Bitplanes— Continued*

Name	***Function***

Subregion Set Plane Mask	`#define prs_putattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

B.6. Rasterops

Table B-6 *Rasterops*

Name	*Function*

Draw Textured or Solid Lines with Width	`#define pr_line(pr, x0, y0, x1, y1, brush, tex, op)`
	`Pixrect *pr;`
	`int x0, y0, x1, y1;`
	`struct pr_brush *brush;`
	`struct pr_texture *tex;`
	`int op;`

Draw Textured Polygon	`pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op,`
	`    spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`int dx, dy`
	`int nbnds, npts[];`
	`struct pr_pos *vlist;`
	`int op, sx, sy;`

Draw Vector	`#define pr_vector(pr, x0, y0, x1, y1, op, value)`
	`Pixrect *pr;`
	`int x0, y0, x1, y1, op, value;`

Get Pixel Value	`#define pr_get(pr, x, y)`
	`Pixrect *pr;`
	`int x, y;`

Masked RasterOp	`#define pr_stencil(dpr, dx, dy, dw, dh, op,`
	`stpr, stx, sty, spr, sx, sy)`
	`Pixrect *dpr, *stpr, *spr;`
	`int dx, dy, dw, dh, op, stx, sty, sx, sy;`

Multiple RasterOp	`#define pr_batchrop(dpr, dx, dy, op, items, n)`
	`Pixrect *dpr;`
	`int dx, dy, op, n;`
	`struct pr_prpos items[];`

RasterOp	`#define pr_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`int dx, dy, dw, dh, op, sx, sy;`

Replicated Source RasterOp	`pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`int dx, dy, dw, dh, op, sx, sy;`

Set Pixel Value	`#define pr_put(pr, x, y, value)`
	`Pixrect *pr;`
	`int x, y, value;`

Table B-6 *Rasterops— Continued*

Name	*Function*

Subregion Draw Vector	`#define prs_vector(pr, pos0, pos1, op, value)`
	`Pixrect *pr;`
	`struct pr_pos pos0, pos1;`
	`int op, value;`

Subregion Get Pixel Value	`#define prs_get(srcprpos)` `struct pr_prpos srcprpos;`

Subregion Masked RasterOp	`#define prs_stencil(dstregion, op, stenprpos, srcprpos)`
	`struct pr_subregion dstregion;`
	`int op;`
	`struct pr_prpos stenprpos, srcprpos;`

Subregion Multiple RasterOp	`#define prs_batchrop(dstpos, op, items, n)`
	`struct pr_prpos dstpos;`
	`int op, n;`
	`struct pr_prpos items[];`

Subregion RasterOp	`#define prs_rop(dstregion, op, srcprpos)`
	`struct pr_subregion dstregion;`
	`int op;`
	`struct pr_prpos srcprpos;`

Subregion Replicated Source RasterOp	`#define prs_replrop(dsubreg, op, sprpos)`
	`struct pr_subregion dsubreg;`
	`struct pr_prpos sprpos;`

Subregion Set Pixel Value	`#define prs_put(dstprpos, value)`
	`struct pr_prpos dstprpos;`
	`int value;`

Trapezon RasterOp	`pr_traprop(dpr, dx, dy, t, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`struct pr_trap t;`
	`int dx, dy, sx, sy op;`

B.7. Double Buffering

Table B-7 *Double Buffering*

Name	*Function*

Get Double Buffering Attributes	`pr_dbl_get(pr, attribute)`
	`Pixrect *pr;`
	`int attribute;`

Set Double Buffering Attributes	`pr_dbl_set(pr, attribute_list)`
	`Pixrect *pr;`
	`int *attribute_list;`

C

Pixrect Data Structures

C

Pixrect Data Structures

Table C-1 *Pixrect Data Structures*

Name	*Data Structure*

Brush	`typedef struct pr_brush {`
	`    int width;`
	`} Pr_brush;`

Character Descriptor	`struct pixchar {`
	`    struct pixrect *pc_pr;`
	`    struct pr_pos pc_home;`
	`    struct pr_pos pc_adv;`
	`};`

Font Descriptor	`typedef struct pixfont {`
	`    struct pr_size pf_defaultsize;`
	`    struct pixchar pf_char[256];`
	`} Pixfont;`

Pixrect	`typedef struct pixrect {`
	`    struct pixrectops *pr_ops;`
	`    struct pr_size pr_size;`
	`    int pr_depth;`
	`    caddr_t pr_data;`
	`} Pixrect;`

Table C-1 *Pixrect Data Structures— Continued*

Name	**Data Structure**

Pixrect Operations	struct pixrectops {
	    int (*pro_rop)();
	    int (*pro_stencil)();
	    int (*pro_batchrop)();
	    int (*pro_nop)();
	    int (*pro_destroy)();
	    int (*pro_get)();
	    int (*pro_put)();
	    int (*pro_vector)();
	    struct pixrect *(*pro_region)();
	#ifdef _PR_IOCTL_DEFINED
	                        0;
	#endif
	    int (*pro_putcolormap)();
	    int (*pro_getcolormap)();
	    int (*pro_putattributes)();
	    int (*pro_getattributes)();
	};

Position	struct pr_pos {
	    int x, y;
	};

Position Within a Pixrect	struct pr_prpos {
	    struct pixrect *pr;
	    struct pr_pos pos;
	};

Size	struct pr_size {
	    int x, y;
	};

Subregion	struct pr_subregion {
	    struct pixrect *pr;
	    struct pr_pos pos;
	    struct pr_size size;
	};

Table C-1 *Pixrect Data Structures— Continued*

<table>
<tr><th>Name</th><th>Data Structure</th></tr>
<tr><td>Texture</td><td><pre>typedef struct pr_texture {
    short *pattern;
    short offset;
    struct pr_texture_options {
        unsigned startpoint : 1,
        endpoint : 1,
        balanced : 1,
        givenpattern : 1,
        res_fat : 1,
        res_poly: 1,
        res_mvlist : 1,
        res_right : 1,
        res_close : 1;
    } options;
    short res_polyoff;
    short res_oldpatln;
    short res_fatoff;
} Pr_texture;</pre></td></tr>
<tr><td>Trapezon</td><td><pre>struct pr_trap {
    struct pr_fall *left, *right;
    int y0, y1;
};</pre></td></tr>
<tr><td>Trapezon Chain</td><td><pre>struct pr_chain {
    struct pr_chain *next;
    struct pr_size size;
    int *bits;
};</pre></td></tr>
<tr><td>Trapezon Fall</td><td><pre>struct pr_fall {
    struct pr_pos pos;
    struct pr_chain *chain;
};</pre></td></tr>
</table>

Index

R

S