Pixrect Reference Manual
Credits and Trademarks

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

SunStation®, Sun Microsystems®, SunCore®, SunWindows®, DVMA®, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

UNIX, UNIX/32V, UNIX System III, and UNIX System V are trademarks of AT&T Bell Laboratories.

Intel® and Multibus® are registered trademarks of Intel Corporation.

DEC®, PDP®, VT®, and VAX® are registered trademarks of Digital Equipment Corporation.

Copyright © 1986 by Sun Microsystems.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.
Contents

Preface .................................................................................................................................................... xi

Chapter 1 Introduction .......................................................................................................... 3
  1.1. Overview .............................................................................................................................. 3
  1.2. Important Concepts ........................................................................................................ 4
  1.3. Example ................................................................................................................................ 5
  1.4. The Pixrect Lint Library .............................................................................................. 5
  1.5. References ............................................................................................................................ 5

Chapter 2 Pixrect Operations ........................................................................................... 9
  2.1. The pixrectops Structure ................................................................................. 10
  2.2. Conventions for Naming Arguments to Pixrect Operations .......... 10
  2.3. Pixrect Errors ..................................................................................................................... 10
  2.4. Creation and Destruction of Pixrects ................................................................. 10
      Create a Primary Display Pixrect ................................................................. 11
      Create Secondary Pixrect ........................................................................... 11
      Release Pixrect Resources ........................................................................... 11
  2.5. Single-Pixel Operations ....................................................................................... 12
      Get Pixel Value .............................................................................................................. 12
      Set Pixel Value ............................................................................................................... 12
  2.6. Constructing an Op Argument ............................................................................. 13
      Specifying a RasterOp Function ........................................................................ 13
      Ops with a Constant Source Value .................................................................. 14
      Controlling Clipping in a RasterOp ................................................................. 14
      Examples of Complete Op Argument Specification .................................. 15
2.7. Multi-Pixel Operations ................................................................. 15
   RasterOp Source to Destination .................................................. 15
   RasterOps through a Mask .......................................................... 16
   Replicating the Source Pixrect .................................................... 16
   Multiple Source to the Same Destination .................................... 17
   Draw Vector .............................................................................. 18
   Draw Textured Polygon .............................................................. 19
2.8. Colormap Access ........................................................................ 22
   Get Colormap Entries .................................................................. 22
   Set Colormap Entries .................................................................. 22
   Inverted Video Pixrects ............................................................... 23
2.9. Attributes for Bitplane Control .................................................... 23
   Get Attributes ........................................................................... 24
   Put Attributes ........................................................................... 24
2.10. Efficiency Considerations .......................................................... 24

Chapter 3 Text Facilities for Pixrects ................................................ 29
  3.1. Pixfonts and Pixchars .............................................................. 29
    Operations on Pixfonts ............................................................. 30
    Load Private Copy of Font ......................................................... 30
    Default Fonts ........................................................................... 30
    Close Font ................................................................................. 31
    Pixrect Text Display ............................................................... 31
    Transparent Text ....................................................................... 31
    Auxiliary Pixfont Procedures .................................................... 31
    Text Bounding Box ................................................................... 32
  3.2. Example .................................................................................. 32

Chapter 4 Memory Pixrects ............................................................... 35
  4.1. The mpr_data Structure .......................................................... 35
  4.2. Creating Memory Pixrects ....................................................... 36
    Create Memory Pixrect ............................................................. 36
    Create Memory Pixrect from an Image ....................................... 36
Example .............................................................................................................................. 36
4.3. Static Memory Pixrects ................................................................................................ 37
4.4. Pixel Layout in Memory Pixrects ............................................................................... 37
4.5. Using Memory Pixrects ............................................................................................... 38

Chapter 5 File I/O Facilities for Pixrects ............................................................................... 41
5.1. Writing and Reading Raster Files ............................................................................. 41
   Write Raster File ........................................................................................................... 41
   Read Raster File ............................................................................................................ 43
5.2. Details of the Raster File Format .............................................................................. 44
5.3. Writing Parts of a Raster File ................................................................................... 45
   Write Header to Raster File .................................................................................... 45
   Initialize Raster File Header ................................................................................... 45
   Write Image Data to Raster File ........................................................................... 46
5.4. Reading Parts of a Raster File ................................................................................... 46
   Read Header from Raster File ............................................................................... 46
   Read Colormap from Raster File ........................................................................... 46
   Read Image from Raster File ................................................................................. 46
   Read Standard Raster File ....................................................................................... 47

Appendix A Writing a Pixrect Driver ................................................................................. 51
A.1. What You'll Need .......................................................................................................... 51
A.2. Implementation Strategy ............................................................................................ 52
A.3. Files Generated ................................................................................................................ 52
   Memory Mapped Devices ............................................................................................ 53
A.4. Pixrect Private Data ....................................................................................................... 53
A.5. Creation and Destruction ............................................................................................ 54
   Creating a Primary Pixrect ...................................................................................... 54
   Creating a Secondary Pixrect ................................................................................... 57
   Destroying a Pixrect .................................................................................................. 58
   The pr_makefun Operations Vector ...................................................................... 58
A.6. Pixrect Kernel Device Driver ..................................................................................... 59
   Configurable Device Support .................................................................................. 59
Open ........................................................................................................................................... 65
Mmap ........................................................................................................................................... 65
ioctl ............................................................................................................................................... 65
Close ........................................................................................................................................... 67
Plugging Your Driver into UNIX ................................................................................................. 67
A.7. Access Utilities .................................................................................................................. 68
A.8. Rop ......................................................................................................................................... 69
A.9. Batchrop .............................................................................................................................. 69
A.10. Vector .................................................................................................................................. 69
Importance of Proper Clipping ................................................................................................. 69
A.11. Colormap ........................................................................................................................... 69
Monochrome .............................................................................................................................. 69
A.12. Attributes ........................................................................................................................... 69
Monochrome .............................................................................................................................. 69
A.13. Pixel .................................................................................................................................... 70
A.14. Stencil ................................................................................................................................ 70
A.15. Curve .................................................................................................................................. 70
A.16. Polygon .............................................................................................................................. 70

Appendix B Pixrect Functions and Macros ............................................................................. 73

Appendix C Pixrect Data Structures ......................................................................................... 81

Appendix D Curved Shapes ......................................................................................................... 85
Tables

Table 2-1 Argument Name Conventions ................................................................. 10
Table 2-2 Useful Combinations of RasterOps ....................................................... 14
Table B-1 Pixrect Library Functions and Macros – Part I .................................... 73
Table B-2 Pixrect Library Functions and Macros – Part II .................................... 75
Table C-1 Pixrect Data Structures ........................................................................ 81
Figures

Figure 1-1  RasterOp Function ............................................................................................... 4
Figure 1-2  Simple Example Program ................................................................................ 5

Figure 2-1  Structure of an  op Argument ........................................................................... 13
Figure 2-2  Example Program with  pr_polygon_2 ............................................................ 20
Figure 2-3  Four Polygons Drawn with  pr_polygon_2 ...................................................... 21

Figure 3-1  Character and  pc_pr Origins .......................................................................... 30
Figure 3-2  Example Program with Text ........................................................................... 32

Figure 4-1  Example Program with Memory Pixrects ....................................................... 37

Figure 5-1  Example Program with  pr_dump ...................................................................... 43
Figure 5-2  Example Program with  pr_load ...................................................................... 44

Figure D-1  Typical Trapezon ............................................................................................ 85
Figure D-2  Some Figures Drawn by  pr_traprop ................................................................. 86
Figure D-3  Trapezon with Clipped Falls .......................................................................... 88
Figure D-4  Example Program with  pr_traprop ................................................................. 89
This document describes the Pixrect graphics library, a low-level RasterOp library for writing device-independent applications for Sun products.

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 is used to indicate file names, function arguments, variables and internal states of Pixrect. Italics are also used in the conventional manner (to emphasize important words and phrases). ALL CAPS is used to indicate values in enumerated types. Bold font is used for the names of Sun software packages. Function names are printed with constant width font.
1

Introduction

Introduction ....................................................................................................................................... 3
1.1. Overview .................................................................................................................................. 3
1.2. Important Concepts ................................................................................................................. 4
1.3. Example ................................................................................................................................... 5
1.4. The Pixrect Lint Library .......................................................................................................... 5
1.5. References ............................................................................................................................... 5
This document describes the Pixrect graphics library, a set of RasterOp routines common among all Sun workstations. With these routines, application programs can be written that access the display on all Sun products.

In the Sun graphics software world, the Pixrect library is a low-level package, sitting on top of the device drivers. For most applications, the higher-level abstractions available in SunView and the Sun graphics libraries are more appropriate.

The Pixrect library is intended only for accessing and manipulating rectangular regions of a display device in a device-independent fashion. There are a few features that are available in higher-level graphics packages like SunView.

Windows
The Pixrect library does not support overlapping window. These can be implemented with memory pixrects by the application, but it is recommended that the functions in SunView be used for this purpose.

Input Devices
The Pixrect library does have any input functions. The application can use input functions in SunView or make calls on the raw input devices (see mouse(4) and keyboard(4).

This document is not a tutorial on writing application programs with the Pixrect library though some simple examples are given. The reader should be familiar with the C programming language and have access to some of the references listed below on bitmap graphics.

1.1. Overview

This manual is divided into chapters that describe the major features of the Pixrect library. Chapter 2 covers the operations for opening and manipulating pixrects. Chapter 3 describes the text facilities in the pixrect library. Chapter 4 discusses memory pixrects rectangular regions of virtual memory that have similar properties to pixrects. Chapter 5 explains the file I/O functions in the Pixrect library. These functions can be used to store and retrieve pixrects from disc files. Appendix A is a implementation guide for pixrect device drivers. Appendix B is a list of the functions and macros in the Pixrect library. Appendix C is a list of types and structures in the Pixrect library. Appendix D describes the curve facilities in Pixrect.
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 from other graphics packages.

Screen Coordinates

The screen coordinate system is two dimensional with the origin in the upper left corner, and $x$ and $y$ increasing to the right and down. The coordinates of a pixel in a pixrect are integers from 0 to the pixrect's width or height minus 1.

Pixels

A pixel is an individual picture element with an address in screen coordinates or relative to some rectangular sub-region of the screen.

Bitmaps

A bitmap is a rectangular region of screen space. Examples of bitmaps include the screen, windows, the cursor or icons.

RasterOps

A RasterOp is an operation involving two or three bitmaps: a source, a destination and a texture. It computes the value of each pixel in the destination bitmap through a boolean operation of the previous value of that destination pixel, of a corresponding source pixel, and possibly a corresponding pixel in a mask. See Chapter 2 for an explanation of the RasterOp functions available in the Pixrect graphics library.

Figure 1-1  RasterOp Function

Pixrects

A pixrect combines the data of a bitmap with operations that can be performed on it. A pixrect can exist on a variety of devices including memory and printers. Since these operations are the same for each device, the programmer does not have to consider the peculiarities of each device when writing an application program.
1.3. Example

The following example draws a line on the display.

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

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

Figure 1-2  Simple Example Program

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

This program can be compiled as follows:

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

This command line compiles the program in line.c. The \texttt{-lpixrect} option causes the C compiler to link the Pixrect library to the application program and create an executable file named line.

The sample program can be executed by the UNIX shell:

```
% line
```

A line will appear in the upper left hand corner of the screen.

1.4. The Pixrect Lint Library

Pixrect provides a \texttt{lint} library which provides type checking beyond the capabilities of the C compiler. For example, you could use the Pixrect \texttt{lint} library to check a program called glass.c with command like this:

```
% lint glass.c -lpixrect
```

Note that most of the error messages generated by \texttt{lint} are warnings, and may not necessarily have any effect on the operation of the program. For a detailed explanation of \texttt{lint}, see the \texttt{lint} chapter in the Programming Tools manual.

1.5. References


[7] *SunCore Reference Manual*.


sun

Version A of 17 February 1986
Pixrect Operations

2.1. The pixrectops Structure ................................................................. 10
2.2. Conventions for Naming Arguments to Pixrect Operations .......... 10
2.3. Pixrect Errors .................................................................................. 10
2.4. Creation and Destruction of Pixrects ............................................... 10
    Create a Primary Display Pixrect ......................................................... 11
    Create Secondary Pixrect .................................................................. 11
    Release Pixrect Resources .................................................................. 11
2.5. Single-Pixel Operations .................................................................. 12
    Get Pixel Value .................................................................................. 12
    Set Pixel Value .................................................................................. 12
2.6. Constructing an Op Argument .......................................................... 13
    Specifying a RasterOp Function .......................................................... 13
    Ops with a Constant Source Value ....................................................... 14
    Controlling Clipping in a RasterOp ...................................................... 14
    Examples of Complete Op Argument Specification ............................ 15
2.7. Multi-Pixel Operations ................................................................... 15
    RasterOp Source to Destination .......................................................... 15
    RasterOps through a Mask .................................................................. 16
    Replicating the Source Pixrect ........................................................... 16
    Multiple Source to the Same Destination .......................................... 17
    Draw Vector ...................................................................................... 18
    Draw Textured Polygon ...................................................................... 19
Pixrect Operations

Pixrect provides procedures to perform the following operations:

- create and destroy a pixrect (open, region and destroy)
- read and write the values of single pixels get( and put)
- use RasterOp functions to affect multiple pixels in a single operation:
  - pr_rrop write from a source pixrect to a destination pixrect,
  - prStencil write from a source pixrect to a destination pixrect under control of a mask,
  - pr_replrop replicate a constant source pixrect pattern throughout a destination pixrect,
  - pr_batchrop write a batch of source pixrects to different locations, in a single destination pixrect
  - pr_vector draw a straight line in a pixrect.
- read and write a colormap (get_colormap, put_colormap)
- select particular bit-planes for manipulation on a color pixrect (get_attributes, put_attributes)

Some of these operations are the same for all pixrecs, and are implemented by a single procedure. These device-independent procedures are called directly by Pixrect clients. Other operations must be implemented differently for each Pixrect device. Each pixrect includes a pointer (in its pr_ops) to a pixrectops structure, that holds the addresses of the particular device-dependent procedures appropriate to that pixrect. This allows clients to access those procedures in a device-independent fashion, by calling the procedure through a pointer, rather than naming the procedure directly. To simplify this indirection, the Pixrect library provides a set of macros which look like simple procedure calls to generic operations, and expand to invocations of the corresponding procedure in the pixrectops structure.

The description of each operation will specify 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, two sets of parallel macros are provided, which differ only in how their
arguments use the geometry data structures.)

2.1. The **pixrectops** Structure

```c
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)());
    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.

2.2. Conventions for Naming Arguments to Pixrect Operations

In general, the conventions listed in Table 2-1 are used in naming the arguments to pixrect operations.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>destination</td>
</tr>
<tr>
<td>s</td>
<td>source</td>
</tr>
<tr>
<td>x and y</td>
<td>left and top origins</td>
</tr>
<tr>
<td>w and h</td>
<td>width and height</td>
</tr>
</tbody>
</table>

The x and y values for functions that operate on pixrects are constrained to be within the boundaries of a pixrect.

2.3. Pixrect Errors

Pixrect procedures which return a pointer to a structure will return NULL when they fail. Otherwise, a return value of PIX_ERR (-1) indicates failure and 0 indicates success. The section describing each library procedure makes note of any exceptions to this convention.

2.4. Creation and Destruction of Pixrects

Pixrects are created by the procedures pr_open and mem_create, by the procedures accessed by the macro pr_region, and at compile-time by the macro mpr_static. Pixrects are destroyed by the procedures accessed by the macro pr_destroy. mem_create and mpr_static are discussed in Chapter 4; the rest of these are described here.
Create a Primary Display

Pixrect

struct 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 by 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 provided that it is a display device, the kernel is configured for it, and it has pixrect support, for example, /dev/bwone0, /dev/bwtwo0, /dev/cgone0 or /dev/cgtwo0.

pr_open does not work for creating a pixrect whose pixels are stored in memory; that function is served by the procedure mem_create, discussed in Chapter 4.

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

Create Secondary Pixrect

#define struct pixrect *pr_region(pr, x, y, w, h)
struct pixrect *pr;
int x, y, w, h;

#define struct pixrect *prs_region(subreg)
struct pr_subregion subreg;

Given an existing pixrect, it is possible to create another pixrect which 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); 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; (x, y) in the existing pixrect will map to (0, 0) in the new one. prs_region does the same thing, but has all its argument values collected into the single structure subreg. Each region procedure returns a pointer to the new pixrect. If it fails, it returns NULL, and prints a message on the standard error output.

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
2.5. Single-Pixel Operations

The next two operations manipulate the value of a single pixel.

Get Pixel Value

#define pr_get(pr, x, y)
struct 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. pr indicates the pixrect in which the pixel is to 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)
struct 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.
2.6. Constructing an Op Argument

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

We describe these three components of the op argument in order.

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

Figure 2-1 Structure of an op Argument

<table>
<thead>
<tr>
<th>color</th>
<th>RasterOp function</th>
<th>clipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Specifying a RasterOp Function

Four bits of the op are used to specify one of the 16 distinct logical functions which 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 2-2.

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

```c
#define PIX_SRC 0x18
#define PIX_DST 0x14
#define PIX_NOT(op) (0x1E & (~op))
```

PIX_SRC and PIX_DST are defined constants, and PIX_NOT is a macro. Together, they allow a desired function to be specified by performing the corresponding logical operations on the appropriate constants. Note that PIX_NOT must be used in all RasterOp operations, and not the ones complement (~) operator.

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 the result:

```c
#define PIX_SET (PIX_SRC | PIX_NOT(PIX_SRC))
```

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

```c
#define PIX_CLR (PIX_SRC & PIX_NOT(PIX_SRC))
```

Other common RasterOp functions are defined in the following table:
Table 2-2  Useful Combinations of RasterOps

<table>
<thead>
<tr>
<th>Op with Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIX_SRC</td>
<td>write (same as source argument)</td>
</tr>
<tr>
<td>PIX_DST</td>
<td>no-op (same as destination argument)</td>
</tr>
<tr>
<td>PIX_SRC</td>
<td>PIX_DST</td>
</tr>
<tr>
<td>PIX_SRC &amp; PIX_DST</td>
<td>mask (AND of source and destination)</td>
</tr>
<tr>
<td>PIX_NOT(PIX_SRC) &amp; PIX_DST</td>
<td>erase (AND destination with negation of source)</td>
</tr>
<tr>
<td>PIX_NOT(PIX_DST)</td>
<td>invert area (negate the existing values)</td>
</tr>
<tr>
<td>PIX_SRC ^ PIX_DST</td>
<td>inverting paint (XOR of source and destination)</td>
</tr>
</tbody>
</table>

Ops with a Constant Source Value

In certain cases, it is desirable to specify an infinite supply of pixels, all with the same value. This is done by using NULL for the source pixrect, and encoding a color in bits 5-31 of the op argument. The following macro supports this encoding:

```c
#define PIX_COLOR(color) ((color)«5)
```

This macro extracts the color from an op:

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

If no color is specified in an op, 0 appears by default. The color specified in the op is used in the case of a null source pixrect or to specify the color of the 'ink' in a monochrome pixrect.

Note that the color is not part of the function component of the op argument; it should never be part of an argument to PIX_NOT.

The color component of op is also used when a monochrome pixrect is written to a color pixrect. In this case:

- if the value of the source pixels = 0, they are painted 0, or background.
- if the value of the source pixels = 1, they are painted color.

If the color component of op is 0 (e.g., because no color was specified), the color will default to (-1) (foreground).

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 which ought to be visible will be 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:

```c
#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 that 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 will specify that source pixels be written to a destination, clipping as they go:

\[ \text{op} = \text{PIX\_SRC}; \]

A more complicated example will be used to affect a rectangle (known to be valid) with a constant red color defined elsewhere. (The function is syntactically correct; it's not clear how useful it is to XOR a constant source with the negation of the OR of the source and destination):

\[ \text{op} = (\text{PIX\_SRC} \otimes \text{PIX\_NOT(PIX\_SRC} \mid \text{PIX\_DST)}) \mid \text{PIX\_COLOR(red)} \mid \text{PIX\_DONTCLIP} \]

2.7. Multi-Pixel Operations

The following operations all apply to 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 the previous section, to specify how pixels are to be set in the destination. Appendix D describes the pr\_traprop curve rendering function.

**RasterOp Source to Destination**

\#define pr\_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
struct pixrect *dpr, *spr;
int dx, dy, dw, dh, op, spr, 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 will be 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) an 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. op specifies the operation which is performed; its construction is described in preceding sections.

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 are smaller than the destination size given. pr\_rop procedures return PIX\_ERR if they fail, 0 if they succeed.
Source and destination pix rects generally must be the same depth. The only exception allows monochrome pix rects to be sources to a destination of any depth. In this case, source pixels = 0 are interpreted as 0 and source pixels = 1 are written as the color value from the op argument. If the color value in the op argument is 0, source pixels = 1 are written as the maximum value which can be stored in a destination pixel.

See the example program in Figure 4-1 for an illustration of pr_rop.

**RasterOps through a Mask**

```c
#define pr_stencil(dpr, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy)
struct 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 pix rect only in areas specified by a third (stencil) pix rect. pr_stencil is identical to pr_rop except that the source pix rect is written through a stencil pix rect which functions as a spatial write-enable mask. The stencil pix rect must be a monochrome memory pix rect. The indicated raster operation is applied only to destination pixels where the stencil pix rect is non-zero. Other destination pixels remain unchanged. The rectangle from (sx, sy) in the source pix rect spr is aligned with the rectangle from (stx, sty) in the stencil pix rect stpr, and written to the rectangle at (dx, dy) with width dw and height dh in the destination pix rect dpr. The source pix rect 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**

```c
pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
struct 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 pix rect. The area affected is described by the rectangle defined by dx, dy, dw, dh. spr indicates the source
Multiple Source to the Same Destination

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 2.6.

The effect of pr replrop is the same as though 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 will have 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. In this case, the source pixel which should be aligned with the destination position is the one which has the same coordinates as that destination pixel, modulo the size of the source pix­rect. pr replrop will perform 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 procedures return PIX_ERR if they fail, 0 if they succeed. Intern­nally 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)
struct 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. 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 2.6, and an array of \texttt{pr\_prpos} addressed by the argument items, and whose length is given in the argument \texttt{n}.

The destination position is initialized to the position given by \texttt{dx} and \texttt{dy}. Then, for each item, the offsets given in \texttt{pos} are added to the previous destination position, and the operation specified by \texttt{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 \texttt{pr\_batchrop} procedures is in painting text; additional facilities to support this application are described in Chapter 3. Note that the definition of \texttt{pr\_batchrop} procedures supports variable-pitch and rotated fonts, and non-roman writing systems, as well as simpler text.

\texttt{pr\_batchrop} procedures return \texttt{PIX\_ERR} if they fail, 0 if they succeed. Internally \texttt{pr\_batchrop} may use \texttt{pr\_rop} procedures. In this case, \texttt{pr\_rop} errors are detected and returned by \texttt{pr\_batchrop}.

### Draw Vector

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

#define prs_vector(pr, pos0, pos1, op, value)
struct pixrect *pr;
struct pr\_prpos pos0, pos1;
int op, value;
```

The \texttt{pr\_vector} and \texttt{prs\_vector} macros invoke device-dependent procedures that draw a vector one unit wide between two points in the indicated pixrect. \texttt{pr\_vector} procedures draw a vector in the pixrect indicated by \texttt{pr}, with endpoints at \((x0, y0)\) and \((x1, y1)\), or at \texttt{pos0} and \texttt{pos1} in the case of \texttt{prs\_vector}. Portions of the vector lying outside the pixrect are clipped as long as \texttt{PIX\_DONTCLIP} is 0 in the \texttt{op} argument. The \texttt{op} argument is constructed as described in Section 2.6, and \texttt{value} specifies the resulting value of pixels in the vector. If the color in \texttt{op} is non-zero, it takes precedence over the \texttt{value} argument.

Any vector that is not vertical, horizontal or 45 degree will contain \textit{jaggies}. This phenomenon, known as \textit{aliasing}, is due to the digital nature of the bitmap screen. It can be visualized by imagining a vertical vector. Displace one endpoint horizontally by a single pixel. The resulting line will have to jog over a pixel at some point in the traversal to the other endpoint. Balancing the vector guarantees that the jog will occur in the middle of the vector. \texttt{pr\_vector} draws \textit{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 will be drawn regardless of how the vector is clipped.

See the example program in Figure 1-2 for an illustration of \texttt{pr\_vector}.
pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy)
struct pixrect *dpr, *spr;
int dx, dy
int nbnds, npts[];
struct pr_pos *vlist;
int op, sx, sy;

pr_polygon_2 draws a polygon in a pixrect. The polygon can have holes. In addition, you can fill it with an image or a texture. This routine is like pr_rop except that nbnds, npts and vlist specify the destination region instead of (dw, dh).

nbnds is the number of individual closed boundaries (vertex lists) in the polygon. For example, the polygon may have one boundary for its exterior shape and several boundaries delimiting interior holes. The boundaries may self intersect or intersect each other. Those pixels having an odd wrapping number are painted. That is, if any line connecting a pixel to infinity crosses an odd number of boundary edges, the pixel will be painted.

Polygons can be wrapped by vectors. To do this, draw the vectors using the same vertices of the polygon as endpoints. The edge of the polygon will match the vector pixel for pixel. Note that vectors are balanced (see pr_vector).
Polygons are semi-open in the sense that on some of the edges, pixels are not drawn where the vector would go. The reason is to allow identical polygons (same size and orientation) to exactly tile the plane with no gaps and no overlaps. This greatly reduces the duplication of pixels drawn when the image contains many small adjacent polygons. In Figure 2-3, the edges AB and DA will be drawn, whereas edges BC and CD will not.
Figure 2-2 Example Program with pr_polygon_2

```c
#include <pixrect/pixrect_hs.h>
#include <stdio.h>
#define CENTERX (1152/2)
#define NULLPR (struct pixrect *) NULL

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

main()
{
    int i, nbnds = 1, npts[1];
    struct pixrect *screen;

    npts[0] = 4;

    screen = pr_open("/dev/fb");
    pr_polygon_2(screen, CENTERX, 100, nbnds, npts, vlist0, PIX_SET, NULLPR, 0, 0);
    for (i=0; i<4; i++)
        pr_vector(screen, (vlist0[i].x + CENTERX), (vlist0[i].y + 100),
                   (vlist0[(i+1)%4].x + CENTERX), (vlist0[(i+1)%4].y + 100), PIX_SET, 1);
    pr_polygon_2(screen, CENTERX, 300, nbnds, npts, vlist1, PIX_SET, NULLPR, 0, 0);
    for (i=0; i<4; i++)
        pr_vector(screen, (vlist1[i].x + CENTERX), (vlist1[i].y + 300),
                   (vlist1[(i+1)%4].x + CENTERX), (vlist1[(i+1)%4].y + 300), PIX_SET, 1);
    pr_polygon_2(screen, CENTERX, 500, nbnds, npts, vlist2, PIX_SET, NULLPR, 0, 0);
    for (i=0; i<4; i++)
        pr_vector(screen, (vlist2[i].x + CENTERX), (vlist2[i].y + 500),
                   (vlist2[(i+1)%4].x + CENTERX), (vlist2[(i+1)%4].y + 500), PIX_SET, 1);
    pr_polygon_2(screen, CENTERX, 700, nbnds, npts, vlist3, PIX_SET, NULLPR, 0, 0);
    for (i=0; i<4; i++)
        pr_vector(screen, (vlist3[i].x + CENTERX), (vlist3[i].y + 700),
                   (vlist3[(i+1)%4].x + CENTERX), (vlist3[(i+1)%4].y + 700), PIX_SET, 1);
    pr_close(screen);
}
```
For each of the nbinds boundaries npts specifies the number of points in the boundary. Hence the npts array is nbinds in length. The vlist contains all of the boundary points for all of the boundaries. The number of points in order are npts[0] + ... + npts[nbinds-1]. pr_polygon_2 joins the last point and first point to close each boundary. A boundary with less than 3 points is an error.

The spr source pixrect fills the interior of the polygon as in pr_rrop. The position sx, sy in spr coordinates coincides with position dx, dy in dpr coordinates. If sx = (-5) and sy = (-10), for example, the source pixrect is positioned at (dx+5, dy+10) in dpr coordinates. pr_polygon_2 clips to both spr and dpr except in the case of NULL spr, where the polygon is filled with the color value in op. The source offset sx, sy is used to superimpose the source image over the polygon. The spr must have depth less than or equal to the depth of dpr. A point (pts[n].x, pts[n].y) in the boundary of a polygon is mapped to (dx + pts[n].x, dy + pts[n].y).
2.8. Colormap Access

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

Get Colormap Entries

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

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

#define prs_putcolormap(pr, index, count, red, green, blue) 
struct 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 will be set to a dark background and light foreground. If the value is non-zero, the foreground will be dark, e.g. black-on-white. Monochrome pixrects are dark-on-light by default.
Inverted Video Pixrects

```
pr_blackonwhite(pr, min, max)
struct pixrect *pr;
int min, max;

pr_whiteonblack(pr, min, max)
struct pixrect *pr;
int min, max;

pr_reversevideo(pr, min, max)
struct 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. These will most often be 0 and 1 for monochrome pixrects; the more general definitions allow colormap-sharing schemes.

"Black-on-white" means that zero (background) pixels will be 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 will be 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, subsequent operations will have inverted intensities. For color frame buffers, the colormap is modified immediately, which affects everything in the display.

2.9. Attributes for Bitplane Control

In a color pixrect, it is often useful to define bitplanes which may 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 Attributes

#define pr_getattributes(pr, planes)
struct pixrect *pr;
int *planes;

#define prs_getattributes(pr, planes)
struct pixrect *pr;
int *planes;

The macros pr_getattributes and prs_getattributes invoke device-dependent procedures that retrieve the mask which controls 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 Attributes

#define pr_putattributes(pr, planes)
struct pixrect *pr;
int *planes;

#define prs_putattributes(pr, planes)
struct pixrect *pr;
int *planes;

The macros pr_putattributes and prs_putattributes invoke device-dependent procedures that manipulate a mask which controls 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 having a value of 1 will be 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.

2.10. 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_batch_rop` cuts down the overhead of painting many small pixrects.
• For small standard shapes `pr_rop` should be used instead of `pr_polygon_2`. 
Text Facilities for Pixrects

3.1. Pixfonts and Pixchars ................................................................. 29
   Operations on Pixfonts ................................................................. 30
   Load Private Copy of Font ............................................................. 30
   Default Fonts .............................................................................. 30
   Close Font .................................................................................. 31
   Pixrect Text Display ................................................................. 31
   Transparent Text ......................................................................... 31
   Auxiliary Pixfont Procedures ...................................................... 31
   Text Bounding Box ...................................................................... 32

3.2. Example .................................................................................... 32
Text Facilities for Pixrects

Displaying text is an important task in many applications, so pixrect-level facilities are provided to address it directly. These facilities fall into two main categories: a standard format for describing fonts and character images, with routines for processing them; and a set of routines which take a string of text and a font, and handle various parts of painting that string in a pixrect.

3.1. Pixfonts and Pixchars

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 will have NULL in their entry in pc_pr. pc_home is the origin of pixrect pc_pr (its upper left corner) relative to the character origin. A character's origin is the leftmost end of its baseline, which is the lowest point on characters without descenders. Figure 3-1 illustrates the pc_pr origin and the character origin.

The leftmost point on a character is normally its origin, but kerning or mandatory letter spacing may move the origin 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 only advances horizontally, rotated fonts may have a vertical advance. Both are provided for in the font.

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 unled interline spacing for that font.)
Note: The definition of a pixfont is expected to change.

Figure 3-1 Character and pc_pr Origins

Operations on Pixfonts

struct 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.

Load Private Copy of Font

struct pixfont *pf_open_private(name)
char *name;

pf_open 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

struct pixfont *pf_default()

The procedure pf_default performs the same 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.
Chapter 3 — Text Facilities for Pixrects

Close Font

```
pf_close(pf)
struct 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` or `pf_default`.

Pixrect Text Display

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

Characters are written into a pixrect with the `pf_text` procedure. The `where` argument is the destination for the start of the text (nominal left edge, baseline; see Section 3.1); `op` is the raster operation to be used in writing the text, as described in Section 2.6; `font` is a pointer to the font in which the text is to be displayed; and `text` is the actual null-terminated string to be displayed. No error indicators are returned. **Note:** The color specified in the `op` specifies the color of the ink. The background of the text is painted 0 (background color).

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

`pf_ttext` paints "transparent" text: it doesn't disturb destination pixels in blank areas of the character's image. The arguments to this procedure are the same as for `pf_text`. The characters' bitmaps are used as a stencil, and the color specified in `op` is painted through the stencil. No error indicators are returned.

(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.)

Transparent Text

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

`pf_ttext` paints "transparent" text: it doesn't disturb destination pixels in blank areas of the character's image. The arguments to this procedure are the same as for `pf_text`. The characters' bitmaps are used as a stencil, and the color specified in `op` is painted through the stencil. No error indicators are returned.

(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;
struct pixfont *font;
char *text;
```

`pf_textbatch` is used internally by `pf_text`; it constructs an array of `pr_prpos` structures and records its length, as required by `batchrop` (see Section 2.7). `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 to be used to display it. When the function returns, `lengthp` will refer 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 which is computed by taking the product of len, is 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;
struct 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.

3.2. Example

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

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

main()
{
    struct pixrect *screen;

    struct pr_prpos where;
    int op = PIX_SET;
    struct pixfont *font;
    char *text = "This is a string."

    screen = pr_open("/dev/fb");
    font = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.l2");

    where.pr = screen;
    where.pos.x = 400;
    where.pos.y = 400;

    pf_ttext(where, op, font, text);

    pf_close(font);
    pr_close(screen);
}
```

Figure 3-2  Example Program with Text
Memory Pixrects

Memory Pixrects ....................................................................................................................................... 35

4.1. The mpr_data Structure .................................................................................................................. 35
4.2. Creating Memory Pixrects .............................................................................................................. 36
    Create Memory Pixrect ..................................................................................................................... 36
    Create Memory Pixrect from an Image ............................................................................................ 36
    Example ............................................................................................................................................. 36
4.3. Static Memory Pixrects .................................................................................................................. 37
4.4. Pixel Layout in Memory Pixrects .................................................................................................... 37
4.5. Using Memory Pixrects .................................................................................................................. 38
Memory Pixrects

4.1. The mpr_data Structure

Memory pixrects store their pixels in memory, instead of displaying them on some display, 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, and the device-dependent operations appropriate to manipulating them are available through their pr_ops. Beyond this, however, the format of the data which describes the particular pixrect is also public: pr_data will hold the address of an mpr_data struct described below. Thus, a client may construct and manipulate memory pixrects using non-pixrect operations. There is also a public procedure, mem_create, which dynamically allocates a new memory pixrect, and a macro, mpr_static, which can be used to generate an initialized memory pixrect in the code of a client program.

```c
struct mpr_data {
    int md_linebytes;
    short *md_image;
    struct pr_pos md_offset;
    short md_primary;
    short md_flags;
};
#define MP_DISPLAY
#define MP_REVERSEVIDEO
```

The pr_data element of a memory pixrect points to an mpr_data struct, which contains the information needed to deal with a memory pixrect.

linebytes is the number of bytes stored in a row of the primary pixrect. This is the difference in the addresses between two pixels at the same x-coordinate, one row apart. Because a secondary pixrect may not include the full width of its primary pixrect, this quantity cannot be computed from the width of the pixrect — see Section 2.4. The actual pixels of a memory pixrect are stored someplace else in memory, usually an array, which md_image points to; the format of that area is described in the next section. The creator of the memory pixrect must ensure that md_image contains an even address. md_offset is the x,y position of the first pixel of this pixrect in the array of pixels addressed by md_image. md_primary is 1 if the pixrect is primary and had its image allocated dynamically (e.g. by mem_create). In this case, md_image will point to an area not referenced by any other primary pixrect. This flag is interrogated by the pr_destroy routine: if it is 1 when that routine is called, the pixrect's
image memory will be freed.

md_flags & (MP_DISPLAY) is non-zero if this memory pixrect is in fact a display device. Otherwise, it is 0. (md_flags & MP_REVERSEVIDEO) is 1 if reversevideo is currently in effect for the display device. md_flags is present to support memory-mapped display devices like the Sun-2 monochrome video device.

Several macros are defined in <pixrect/memvar.h> to aid in addressing memory pixrects. The following macro obtains a pointer to the mpr_data of a memory pixrect.

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

The following macro computes the bytes per line of a primary memory pixrect given its width in pixels and the bits per pixel. This includes the padding to word bounds. It is useful for incrementing pixel addresses in the y direction.

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

4.2. Creating Memory Pixrects

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

Create Memory Pixrect

struct pixrect *mem_create(w, h, depth)
int w, h, depth;

A new primary pixrect is created by a call to the procedure 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, and a pointer to the pixrect is returned. If this can not be done, the return value is NULL.

Create Memory Pixrect from an Image

struct pixrect *mem_point(width, height, depth, data)
int width, height, depth;
short *data;

The mem_point routine builds a pixrect structure that points to a dynamically created image in memory. Client programs may use this routine 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 to be associated with the pixrect.

Example

Here is an example program program that uses memory pixrects to make an inverted copy of the frame buffer. It opens the default frame buffer and two memory pixrects, one the size of a scan line and the other the size of the frame buffer. It then copies in reverse order the frame buffer line by line into the larger memory pixrect. Finally it copies the memory pixrect back into the frame buffer.
4.3. Static Memory Pixrects

A memory pixrect may be created at compile time by using the `mpr_static` macro. `name` is a 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 an even-byte aligned data object that contains the pixel values in the format described above.

The macro generates two structures:

```c
struct mpr_data name_data;
struct pixrect name;
```

The `mpr_data` is initialized to point to all of the image data passed in; the pixrect then refers to `mem_ops` and to `name_data`. **Note:** Contrary to its name, this macro generates structures of storage class `extern`.

4.4. Pixel Layout in Memory Pixrects

In memory, the upper-left corner pixel is stored at the lowest address. This address must be even. That first pixel is followed by the remaining pixels in the top row, left-to-right. Pixels are stored in successive bits without padding or alignment. For pixels more than 1 bit deep, it is possible for a pixel to cross a byte boundary. However, rows are rounded up to 16-bit boundaries. After any padding for the top row, pixels for the row below are stored, and so on through the whole rectangle. Currently, memory pixrects are only supported for pixels of...
1, 8, 16, or 24 bits. If source and destination are both memory pixrects they must have an equal number of bits per pixel.

4.5. Using Memory Pixrects

Memory pixrects can be used to get data from and send data to the display device. Several routines exist for interfacing Pixwins with memory pixrects. These include `pw_read`, `pw_rop` and `pw_write`. Refer to the SunView Programmer’s Guide for more details. For applications using the raw device without SunView, `pr_rop` can be used for operations on memory pixrects.

Another use of memory pixrects is for processing images that 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 5.
File I/O Facilities for Pixrects

5.1. Writing and Reading Raster Files ................................................................. 41
   Write Raster File ............................................................................................ 41
   Read Raster File ............................................................................................ 43

5.2. Details of the Raster File Format .................................................................. 44

5.3. Writing Parts of a Raster File ....................................................................... 45
   Write Header to Raster File ......................................................................... 45
   Initialize Raster File Header ....................................................................... 45
   Write Image Data to Raster File ................................................................... 46

5.4. Reading Parts of a Raster File ...................................................................... 46
   Read Header from Raster File ...................................................................... 46
   Read Colormap from Raster File ................................................................. 46
   Read Image from Raster File ....................................................................... 46
   Read Standard Raster File ....................................................................... 47
5.1. Writing and Reading Raster Files

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

Write Raster File

```c
int pr_dump(input_pr, output, colormap, type, copy_flag)
struct 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 `input_pr` pixrect can be a secondary pixrect. This allows the caller to 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 desired output type should either be one of the following standard types or correspond to a customer provided filter.

```c
#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 encoding of bytes of the pixrect image; usually this results in shorter files. Specifying any other output type causes pr_dump to pipe a raster file of RT_STANDARD type to the filter named /usr/lib/rasfilters/convert.type, where type is the ASCII corresponding to the specified type in decimal. The output of the filter is then copied to output.

It is strongly recommended that customer-defined formats use a type of 100 or more, to avoid conflicts with additions to the set of standard types. To aid in development of filters for customer-defined formats, pr_dump recognizes the RT_EXPERIMENTAL type as special, and uses the filter named ./convert.65535 rather than /usr/lib/rasfilters/convert.65535.

#define RT_EXPERIMENTAL 65535

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 not monochrome, pr_dump will attempt to write the colormap obtained from input_pr (via pr_getcolormap assuming a 256 element RGB colormap). The following structure is used to specify 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

If the colormap type is RMT_NONE, then the colormap length must be 0. This case usually arises when dealing with monochrome displays and monochrome 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. For developers of customer-defined formats, the following colormap type is provided but not interpreted by the pixrect software:

#define RMT_RAW 2

Finally copy_flag specifies whether or not input_pr should be copied to a temporary pixrect before the image is output. There are two situations in which the copy_flag value should be non-zero:

- if the output type is RT_BYTE_ENCODED — This is because the encoding algorithm does the encoding in place and will destroy the image data of input_pr if it fails while working on input_pr directly.
- if input_pr is a pixrect in a frame buffer that is likely to be asynchronously modified — Note that use of copy_flag will still not guarantee that the correct image will be output unless the pr_rrop to copy from the frame buffer
is atomic or otherwise made uninterruptable.

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

#define FALSE 0
#define TRUE !FALSE

main()
{
    struct pixrect *screen, *icon;
    FILE *output = stdout;
    colormap_t *colormap = NULL;
    int type = RT_STANDARD;
    int copy_flag = TRUE;

    screen = pr_open("/dev/fb");
    icon = pr_region(screen, 1050, 10, 64, 64);
    pr_dump(icon, output, colormap, type, copy_flag);
    pr_close(screen);
}
```

Figure 5-1 Example Program with `pr_dump`

Read Raster File

```c
struct pixrect *pr_load(input, colormap)
FILE *input;
colormap_t *colormap;

The `pr_load` can be used to retrieve the image described by a 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 problems occur, `pr_load` returns `NULL` instead.

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` will simply discard any and all colormap information contained in the specified input raster file.
#include <pixrect/pixrect_hs.h>
#include <stdio.h>

main()
{
    struct pixrect *screen, *icon, *pr_load();
    FILE *input = stdin;
    colormap_t *colormap = NULL;

    screen = pr_open("/dev/fb");
    icon = pr_load(input, colormap);
    pr_rop(screen, 1050, 110, 64, 64, PIX_SET, icon, 0, 0);
    pr_close(screen);
}

Figure 5-2  Example Program with  pr_load

5.2. 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 8 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 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 the memory 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,
5.3. 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

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

For clients that do not want to explicitly initialize the rasterfile struct the following 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 will be 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, the returned pixrect is simply input_pr. However, if copy_flag is non-zero, the returned pixrect is dynamically allocated and the caller is responsible for deallocating the returned pixrect after it is no longer needed.
Write Image Data to Raster File

```c
text
int pr_dump_image(pr, output, rh)
struct 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 returns PIX_ERR.

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

5.4. 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

```c
text
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 returns PIX_ERR.

Read Colormap from Raster File

```c
text
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, they can be read by calling `pr_load_colormap`. If the specified colormap is NULL, this routine will skip over the colormap values by reading and discarding them. Note that the caller is responsible for looking at the raster file header and setting up an appropriate colormap struct before calling this routine.

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

Read Image from Raster File

```c
text
struct 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 rasterfile and the colormap structures will be modified as a side-effect 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 instead of a pixrect containing the image.
Read Standard Raster File

```c
struct 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 will not invoke a filter on non-standard raster file types.
Writing a Pixrect Driver

A.1. What You’ll Need .......................................................................................................... 51
A.2. Implementation Strategy ............................................................................................ 52
A.3. Files Generated ................................................................................................................ 52
  Memory Mapped Devices ........................................................................................ 53
A.4. Pixrect Private Data ...................................................................................................... 53
A.5. Creation and Destruction ........................................................................................... 54
  Creating a Primary Pixrect ..................................................................................... 54
  Creating a Secondary Pixrect .............................................................................. 57
  Destroying a Pixrect ............................................................................................. 58
  The pr_makefun Operations Vector ..................................................................... 58
A.6. Pixrect Kernel Device Driver .................................................................................. 59
  Configurable Device Support ............................................................................. 59
  Open ..................................................................................................................... 65
  Mmap ..................................................................................................................... 65
  ioctl ......................................................................................................................... 65
  Close ...................................................................................................................... 67
  Plugging Your Driver into UNIX ............................................................................ 67
A.7. Access Utilities ................................................................................................................ 68
A.8. Rop .......................................................................................................................................... 69
A.9. Batchrop ............................................................................................................................... 69
A.10. Vector ............................................................................................................................... 69
    Importance of Proper Clipping ............................................................................. 69
A.11. Colormap ............................................................ 69
    Monochrome ............................................................. 69
A.12. Attributes ............................................................ 69
    Monochrome ............................................................. 69
A.13. Pixel ................................................................. 70
A.14. Stencil ................................................................. 70
A.15. Curve ................................................................. 70
A.16. Polygon ............................................................... 70
Writing a Pixrect Driver

Sun has defined a common programming interface to pixel addressable devices that enables, in particular, 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 describes how to write a pixrect driver. It is assumed that you have already read Chapter 2; it 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 for the Sun Workstation. The section in that manual on writing the kernel device driver portion of the pixrect implementation is important.

This chapter contains auxiliary material of interest only to pixrect driver implementors, not programmers accessing the pixrect interface. This document explains how to plug a new pixrect driver into the software architecture so that it may be used in a device independent manner. Also, utilities and conventions that may be of use to the pixrect driver implementor are discussed.

This chapter walks through some of the C language source code for the pixrect driver for the Sun-1 color frame buffer. There is no significance to the fact that we are using the Sun-1 color frame buffer as an example. Another pixrect driver would have been just as good.

The actual source code that is presented here is boiler-plate, i.e., almost every pixrect driver will be the same. You should be able to make your own driver just from the documentation alone. However, 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 is available as a source code purchase option (available without a UNIX source license).

A1. What You’ll Need

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

- You need the correct documentation:

A.2. 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) a physical device from a user process. The private cgl_make routine must be written. The cgl_region and cgl_destroy pixrect operation must be written.

• Write and debug the basic pixel rectangular region function. The cgl_putattributes and cgl_put colormap pixrect operations must be written in addition to the cgl_rop routine.

• Write and debug batchrop routines. The cgl_batchrop pixrect operation must be written.

• Write and debug vector drawer. The cgl_vector pixrect operation must be written.

• Write and debug remaining pixrect operations: cgl_stencil, cgl_get, cgl_put, cgl_getattributes and cgl_get colormap.

• 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 new pixrect driver. Experience has shown that when you are able to run released SunView programs that your pixrect driver is in pretty good shape.

A.3. Files Generated

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

• cglreg.h - A header file describing the structure of the raster device. It contains macros used to address the raw device.

• cgivar.h - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.


• 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. (One could imagine a device that doesn't even meet the minimum requirements being used as a pixel addressable device. We will not discuss any of the ways that such a device might fake 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.
Memory Mapped Devices

Some devices are memory mapped, e.g., the Sun-2 monochrome video frame buffer. With such devices, their pixels are manipulated directly as main memory; there are no device specific registers through which the pixels are accessed. Memory mapped devices are able to rely on the memory pixrect driver for many of its operations. The only files that the Sun 2 monochrome video frame buffer supplies are:

- bw2var.h - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.
- /sys/sundeve/bwtwo.c - The pixrect kernel device driver code.
- bw2.c - The pixrect creation and destruction routines.

The operations vector for the Sun 2 monochrome pixrect driver is:

```c
struct pixrectops bw2_ops = {
    mem_rop, mem_stencil, mem_batchrop,
    0, bw2_destroy, mem_get, mem_put, mem_vector,
    mem_region, mem_putcolormap, mem_getcolormap,
    mem_putattributes, mem_getattributes
};
```

A.4. Pixrect Private Data

Each pixrect device must have a private data object that contains instance specific data about the state of the driver. It is not acceptable to have global data shared among all the pixrects objects. The device specific portion of the pixrect data must contain certain information:

- An offset from the upper left-hand corner of the pixel device. This offset, plus the width and height of the pixrect from the public portion, is used to determine the clipping rectangle used during pixrect operations.
• A flag for distinguishing between primary and secondary pixrects. Primary pixrects are the owners of dynamically allocated resources shared between primary and secondary pixrects.

• A file descriptor to the pixrect kernel device. Usually, the file descriptor is used while mapping pages into the user process address space so that the device may be addressed. One could imagine a pixrect driver that had some of its pixrect operations implemented inside the kernel. The file descriptor would then be the key to communicating with that portion of the package via read(2), write(2) and ioctl(2) system calls.

Here is other possible data maintained in the pixrect’s private data:

• For many devices, a virtual address pointer is part of the private data so that the device can be accessed from user code.
• For color devices, there is a mask to enable access to specific bit planes.
• For monochrome devices, there is a video invert flag. This replaces the color-map of color devices.

A.5. Creation and Destruction

This section covers the code for pixrect object creation and destruction. Code for the Sun-1 color frame buffer pixrect driver is presented as an example.

There are three public pathways to creating a pixrect:

• pr_open creates a primary pixrect.
• pr_region creates a secondary pixrect which specifies a subregion in an existing pixrect.

There are two public pathways to destroying a pixrect:

• pr_destroy destroys a primary or secondary pixrect. Clients of the pixrect interface are responsible for destroying all extant secondary pixrects before destroying the primary pixrect from which they were derived.
• pr_close simply calls pr_destroy.

Creating a Primary Pixrect

In this section, the private cg1_make pixrect operation is described. This is the flow of control for pr_open:

• Higher levels of software call pr_open, which takes a device file name (e.g., /dev/cgone0).
• pr_open opens the device and finds out its type and size via an FBIOGTYPE ioctl(2) call (see <sun/fbio.h>).
• pr_open uses the type of pixel addressable device to index into the pr_makefun array of procedures (more on this later) and calls the referenced pixrect make function, cg1_make.
• cg1_make returns the primary pixrect (it workings are discussed below).
• pr_open closes its handle on the device and the pixrect is returned.
Here is a partial listing of `cgl.c` that contains code that is important to the `cgl_make` procedure. As it is for other code presented in this document, it is here so you can refer back to it as you read the subsequent explanation. Some lines are numbered for reference and normal C comments have been removed in favor of the accompanying text.

```c
#include <sys/types.h>
#include <stdio.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_util.h>
#include <pixrect/cglreg.h>
#include <pixrect/cglvar.h>

static struct pr_devdata *cgldevdata; /* cgl.1*/

struct pixrectops cgl_ops = { /* cgl.2*/
    cgl_rop, cglStencil, cglBatchrop, 0, cgl_destroy, cgl_get,
    cgl_put, cgl_vector, cgl_region, cgl_putcolormap, cgl_getcolormap,
    cgl_putattributes, cgl_getattributes,
};

struct pixrect *
cgl_make(fd, size, depth) /* cgl.3*/
    int fd; /* cgl.4*/
    struct pr_size size;
    int depth;
{
    struct pixrect *pr;
    register struct cglpr *cgpr; /* cgl.5*/
    struct pr_devdata *dd; /* cgl.6*/

    if (depth != CG1_DEPTH || size.x != CG1_WIDTH
        || size.y != CG1_HEIGHT) { /* cgl.7*/
        fprintf(stderr, "cgl_make sizes wrong %D %D %D\n",
            depth, size.x, size.y);
        return (0);
    }

    if (! (pr = pr_makefrornfd(fd, size, depth, &cgldevdata, &dd,/* cgl.8*/
            sizeof(struct cglfb), sizeof(struct cglpr), 0))
            sizeof(struct cglfb), sizeof(struct cglpr), 0))
        return (0);
    pr->pr_ops = &cgl_ops; /* cgl.9*/
    cgpr = (struct cglpr *)pr->pr_data; /* cgl.10*/
    cgpr->cgpr_fd = dd->fd; /* cgl.11*/
    cgpr->cgpr_va = (struct cglfb *)dd->va; /* cgl.12*/
    cgpr->cgpr_planes = 255; /* cgl.13*/
    cgpr->cgpr_offset.x = cgpr->cgpr_offset.y = 0; /* cgl.14*/
    cgl_setreg(cgpr->cgpr_va, CG_STATUS, CG_VIDEOENABLE); /* cgl.15*/
    return (pr); /* cgl.16*/
}
```

Version A of 17 February 1986
Line *cg*1.7 does some consistency checking to make sure that the dimensions of the pixel addressable device and the client’s idea about the dimensions of the device match.

```c
struct *pixrect pr_makefromfd(fd, size, depth, devdata, curd;
  mmapbytes, privdatabytes, mmapoffsetbytes)
struct pr_size size;
struct pr_devdata **devdata, **curdd;
int fd, depth, mmapbytes, privdatabytes, mmapoffsetbytes;
```

Line *cg*1.8 calls the pixrect library routine *pr_makefromfd* to do most of the work:

- Allocates a pixrect structure object using the calloc library call. The pixrect is filled in with *size* and *depth* parameters.
- Allocates an object of the size *privdatabytes* using the calloc library call and placing a pointer to it in the pr_data field of the allocated pixrect.
- *dup(2)*s the passed in file descriptor *fd* so that when the caller closes the file descriptor the device wouldn’t close.
- *valloc(2)*s the amount of space *mmapbytes*.
- *mmap(2)*s the space returned from vallob to the device.
- If an error is detected during any of the above calls, an error is written to the standard error output. A NULL pixrect handle is returned in this case.
- Returns the allocated pixrect.

This brings us to the issue of minimizing resources used by the pixrect driver. *andpr_open, cg1_make*, can be (and are) called many times thus creating a situation in which there are many primary pixrects open at a time. A pixrect should maintain an open file descriptor and (usually) a non-trivial amount of virtual address space mapped into the user process’s address space. Both the number of open file descriptors, the virtual address space (max 16 megabytes) and the disk swap space needed to support the virtual memory (configurable) are finite resources. However, multiple open pixrects can share all these resources.

The pixrect library supports a resource sharing mechanism, part of which is implemented in *pr_makefromfd*. The devdata parameter passed to *pr_makefromfd* is the head of a linked list of pr_devdata structures of which there is one per pixrect driver. It is sufficient to say that through the data maintained on this list, sharing of the scarce resources described above can be accomplished.

The *curdd* parameter passed to *pr_makefromfd* is set to be the pr_devdata structure that applies to the device identified by *fd*.

Lines *cg*1.9 through *cg*1.14 are concerned with initializing the pixrect’s private data with dynamic information described in *dd* (*curdd* in the previous paragraph) and static information about the pixel addressable device.
Creating a Secondary Pixrect

In this section, the `cgl_region` pixrect operation is described. Here is all of `cgl_region.c`.

```c
struct pixrect *cgl_region(src)
    struct pr_subregion src;
{
    register struct pixrect *pr;
    register struct cglpr *scgpr = cgl_d(src.pr), *cgpr;
    int zero = 0;
    pr_clip(&src, &zero); /* cgl_region.1*/
    if ((pr = (struct pixrect *)calloc(1, sizeof (struct pixrect))) == 0) /* cgl_region.2*/
        return (0);
    if ((cgpr = (struct cglpr *)calloc(1, sizeof (struct cglpr))) == 0) /* cgl_region.3*/
        free(pr);
        return (0);
}
pr->pr_ops = &cgl_ops; /* cgl_region.4*/
pr->pr_size = src.size; /* cgl_region.5*/
pr->pr_depth = CGl_DEPTH; /* cgl_region.6*/
pr->pr_data = (caddr_t)cgpr; /* cgl_region.7*/
cgpr->cgpr_fd = -1; /* cgl_region.8*/
cgpr->cgpr_va = scgpr->cgpr_va; /* cgl_region.9*/
cgpr->cgpr_planes = scgpr->cgpr_planes; /* cgl_region.10*/
cgpr->cgpr_offset.x = scgpr->cgpr_offset.x + src.pos.x; /* cgl_region.11*/
cgpr->cgpr_offset.y = scgpr->cgpr_offset.y + src.pos.y; /* cgl_region.12*/
return (pr);
```

cgl_region is less complex than cgl_make. The first thing done is to clip the requested subregion to fall within the source pixrect (line `cgl_region.1`).

Next, objects are allocated for the pixrect and the pixel addressable device’s private data (line `cgl_region.2` and `cgl_region.3`). Then, similarly to the later part of cgl_make, the two new data objects are initialized (lines `cgl_region.4` through `cgl_region.12`). One thing to note is that the cgl driver uses a -1 in the file descriptor field of the pixrect’s private data to indicate that this pixrect is secondary (line `cgl_region.8`).

Line `cgl.15` is where the video signal for the device is enabled. By convention, every raster device should make sure that it is enabled.
Destroying a Pixrect

In this section, the cgl_destroy pixrect operation is described. It works on secondary and primary pixrects. Here is more of cgl.c.

cgl_destroy(pr)
    struct pixrect *pr;
{
    register struct cglpr *cgpr;
    if (pr == 0)
        return (0);
    if (cgpr = cgl_d(pr)) { /*cgl.30*/
        if (cgpr->cgpr_fd != -1) { /*cgl.31*/
            pr_unmakefromfd(cgpr->cgpr_fd, &cg1devdata); /*cgl.32*/
        }
        free(cgpr); /*cgl.33*/
    }
    free(pr); /*cgl.34*/
    return (0);
}

Note that dynamic memory is freed (lines cgl.33 and cgl.34). Also, note that only a primary pixrect (as indicated by a file descriptor that is not -1) invokes a call to pr_unmakefromfd (line cgl.32).

pr_unmakefromfd(fd, devdata)
struct pr_devdata **devdata;
int fd;

This pixrect library routine is the counterpart of pr_makefromfd. If the device identified by the file descriptor fd has no more pixrects associated with it (as determined from devdata) then the resources associated with it are released.

The pr_makefun Operations Vector

As mentioned above, pr_open calls cgl_make through the pr_makefun procedure vector. This is what pr_makefun looks like (it is the sole contents of pr_makefun.c):

```
#include <pixrect/pixrect_hs.h>
#include <sun/fbio.h>
#include <sys/ioctl.h>

struct pixrect *((pr_makefun[FBTYPE_LASTPLUSONE]))() = { 
    (struct pixrect *((*))bw1_make,
    (struct pixrect *((*))cgl1_make,
    (struct pixrect *((*))bw2_make,
    (struct pixrect *((*))cg2_make,
    (struct pixrect *((*))gp1_make,
};
```

pr_makefun is the routine that pulls in all the code from the different frame buffers. If a site is not going to use programs on more than one kind of display, the unused slots can be commented out to prevent the code for the unused display
from being loaded. This has the advantage of reducing disk space usage. However, working set size will presumably not be affected due to virtual memory not touching unused code.

For both the case of adding and deleting drivers, loading a compiled version of this edited file will have the effect of ignoring the commented out device drivers.

When adding some new pixrect driver, you need to assign it some unused constant from `<sun/fbio.h>`, e.g., `FBTYPE_NOTSUN1`. This then becomes the device identifier for your new pixrect driver. You need to generate a new version of the source file `pr_makefun.c` with the above data structure except that the array entry `pr_makefun [FBTYPE_NOTSUN1]` would contain the pixrect make procedure for your `FBTYPE_NOTSUN1` pixrect driver (line `pr_makefun 1`). The old `pr_makefun.o` in the pixrect library could be replaced with your new `pr_makefun.o` using `ar(l)`.

A.6. Pixrect Kernel Device Driver

A pixrect kernel device driver supports the pixel addressable device as a complete UNIX device. It also supports use of this device by the SunView driver so that the cursor can be tracked and the colormap loaded within the kernel. The document *Writing Device Drivers for the Sun Workstation* contains the details of device driver construction. It also contains an overview.

The code in this section comes from `cgone.c`. In the kernel, suffixes that end with a number (like `cg1`) confuse the conventions surrounding device driver names. A number suffix refers to the minor device number of a device. Therefore, in our example, `cg1` becomes `cgone` where the naming has something to do with the pixrect kernel device driver.

Configurable Device Support

Raster devices typically hang off a high speed bus (e.g., Multibus) or are plugged into a high speed communications port. At kernel building time the UNIX auto-configuration mechanism is told what devices to expect and where they should be found. At boot time the auto-configuration mechanism checks to see if each of the devices it expects are present.

This section deals with the auto-configuration aspects of the driver. This driver is written in the conventional style that supports multiple units of the same device type. It is recommended that you follow this style even if you aren’t anticipating multiple pixel addressable devices of your type on a single UNIX system.

```c
#include "cgone.h"
#include "win.h"
#if NCGONE > 0
#include ".h/param.h"
#include ".h/systm.h"
#include ".h/dir.h"
#include ".h/user.h"
#include ".h/proc.h"
#include ".h/buf.h"
#include ".h/conf.h"
#include ".h/file.h"
#include ".h/uio.h"
#include ".h/ioctl.h"
```
#include "../machine/mmu.h"
#include "../machine/pte.h"
#include "../sun/fbio.h"
#include "../sundev/mbvar.h"
#include "../pixrect/pixrect.h"
#include "../pixrect/pr_util.h"
#include "../pixrect/cglreg.h"
#include "../pixrect/cgivar.h"

#if NWIN > 0
#define CG1_OPS &cgl_ops
struct pixrectops cgl_ops = {
    cgl_rop,
    cgl_putcolormap,
    cgl_putattributes,
};
#else
#define CG1_OPS (struct pixrectops *)0
#endif

#define CG1SIZE (sizeof (struct cglfb))
struct cglpr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };

/*
 * Driver information for auto-configuration stuff.
 */
int cgoneprobe(), cgoneintr();
struct pixrect cgonepixrect[NCGONE];
struct cglpr cgoneprdata[NCGONE];
struct mb_device *cgoneinfo[NCGONE];
struct mb_driver cgonedriver = {
    cgoneprobe, 0, 0, 0, 0, cgoneintr,
    CG1SIZE, "cgone", cgoneinfo, 0, 0, 0,
};

/*
 * Only allow opens for writing or reading and writing
 * because reading is nonsensical.
 */
cgoneopen(dev, flag)
    dev_t dev;
{
    return(fbopen(dev, flag, NCGONE, cgoneinfo));
}

/*
 * When close driver destroy pixrect.
 */
/*ARGSUSED*/
Appendix A — Writing a Pixrect Driver

Appendix A - Writing a Pixrect Driver

```c
void cgoneclose(dev, flag)
{
    dev_t dev;
    register int unit = minor(dev);

    if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
        bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cg1pr);
        bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
    }
}

/*ARGSUSED*/
void cgoneioctl(dev, cmd, data, flag)
{
    dev_t dev;
    caddr_t data;
    register int unit = minor(dev);
    switch (cmd) {
    case FBIOGTYPE: {
        register struct fbtype *fb = (struct fbtype *)data;

        fb->fb_type = FBTYPE_SUN1COLOR;
        fb->fb_height = 480;
        fb->fb_width = 640;
        fb->fb_depth = 8;
        fb->fb_cmsize = 256;
        fb->fb_size = 512*640;
        break;
    }
    case FBIOGPIXRECT: {
        register struct fbpixrect *fbpr = (struct fbpixrect *)data;
        register struct cg1fb *cg1fb =
            (struct cg1fb *)cgoneinfo[(unit)]->md_addr;

        /* "Allocate" and initialize pixrect data with default. */
        fbpr->fbpr_pixrect = &cgonepixrect[unit];
        cgonepixrect[unit] = cgonepixrectdefault;
        fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];
        cgoneprdata[unit] = cgoneprdatadefault;

        /* Fixup pixrect data. */
        cgoneprdata[unit].cgpr_va = cg1fb;

        /* Enable video */
        cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);

        /* Clear interrupt */
    }
}
```

Version A of 17 February 1986
We need to handle vertical retrace interrupts here. The color map(s) can only be loaded during vertical retrace; we should put in ioctls for this to synchronize with the interrupts. FOR NOW, see comments in the code.

```c
int cgoneintr()
{
    return (fbintr(NCGONE, cgoneinfo, cgoneintclear));
}

/*ARGSUSED*/
cgonemmap(dev, off, prot)
{
    dev_t dev;
    off_t off;
    int prot;
    return (fbmmap(dev, off, prot, NCGONE, cgoneinfo, CGlSIZE));
}
```

#include ".:/sundev/cgreg.h"

/* Note: using old cgreg.h to peek and poke for now.
We determine that the thing we’re addressing is a color board by setting it up to invert the bits we write and then writing and reading back DATAl, making sure to deal with FIFOs going and coming.

```c
/*ARGSUSED*/

#define DATAl 0x5C
#define DATA2 0x33

int cgoneprobe(reg, unit)
    reg, unit;
{
    register caddr_t CGXBase;
    register u_char *xaddr, *yaddr;

    CGXBase = reg;
    if (pokec((caddr_t)GR_freg, GR_copy_invert))
        return (0);
    if (pokec((caddr_t)GR_mask, 0))
        return (0);
    xaddr = (u_char *) (CGXBase + GR_x_select + GR_update + GR_set0);
    yaddr = (u_char *) (CGXBase + GR_y_select + GR_set0);
    if (pokec((caddr_t)yaddr, 0))
        return (0);
    if (pokec((caddr_t)xaddr, DATAl))
        return (0);
    (void) peekc((caddr_t)xaddr);    (void) pokec((caddr_t)xaddr, DATA2);
    if (peekc((caddr_t)xaddr) == (~DATAl & 0xFF)) {
        /*
         * The Sun-1 color frame buffer doesn't indicate that an interrupt is pending on itself.
         * Also, the interrupt level is user program changeable.
         * Thus, the kernel never knows what level to expect an interrupt on this device and doesn't know if an interrupt is pending.
         * So, we add the cgoneintr routine to a list of interrupt handlers that are called if no one handles an interrupt.
         * Add_default_intr screens out multiple calls with the same interrupt procedure.
         */
        add_default_intr(cgoneintr);
        return (CG1SIZE);
    }
    return (0);
}

#endif
```

This is how the driver is plugged into the auto-configuration mechanism. `/etc/config` reads a line in the configuration file for a Sun-1 color frame.
buffer:

device cgone0 at mb0 csr 0xec000 priority 3

An external reference to cgonedriver (line cgone.4) is made in a table maintained by the auto-configuration mechanism. At boot time, if the auto-configuration mechanism can resolve the reference to cgonedriver then the contents of this structure are used to configure in the device:

- cgoneprobe - The name of the probe procedure (line cgone.5).
- cgoneintr - The name of the interrupt procedure (line cgone.6).
- CGISIZE - The size in bytes of the address space of the device.
- cgone - The prefix of the device. Used in status and error messages.
- cgoneinfo - The array of devices pointers of the driver’s type (line cgone.2).

- The other field’s defaults suffice for most pixel addressable devices.

cgoneprobe is called to let the driver decide if the virtual address at reg is indeed a device that this driver recognizes as one of its own. The unit argument is the minor device number of this device. Writing a good probe routine can be difficult. The trick is to use some idiosyncrasy of the device that differentiates it from others. The real driver for the Sun-1 color frame buffer determines that it is addressing a Sun-1 color frame buffer by setting it up to invert the data written to it and reading back the result. The details of this code are not important to this discussion and is not included. Zero is returned if the probe fails and CGISIZE is returned if the probe succeeds.

cgoneintr is called when an interrupt is generated at the beginning of the vertical retrace. There are a variety of things that one might want to synchronize with such an interrupt, e.g., load the colormap or move the cursor. Currently, the utility fbintr simply disables the interrupt from happening again (line cgone.6).

int fbintr(numdevs, mb_devs, intclear)
    int numdevs;
    struct mb_device **mb_devs;
    int (*intclear)();

numdevs is the maximum number of devices of this type configured. mb_devs is the array of devices descriptions. intclear is called back to actually turn off the interrupt for a particular device. intclear must have the same calling sequence as cgoneintclear (line cgone.7), i.e., it take the virtual address of the device to disable interrupts. cgl_intclear (line cgone.8) is a macro that actually disables the interrupts of cglfb.
Open

When an open system call is made at the user level cgoneopen is called.

```c
int cgoneopen(dev, flag)
    dev_t dev;
    {
        return(fbopen(dev, flag, NCGONE, cgoneinfo));
    }
```

cgoneopen uses the utility fbopen.

```c
int fbopen(dev, flag, numdevs, mb_devs)
    dev_t dev;
    int flag, numdevs;
    struct mb_device **mb_devs;
```

fbopen checks to see if dev is available for opening. If not the error ENXIO is returned. If flag doesn’t ask for write position (FWRITE) then the error EINVAL is returned. Normally, zero is returned on a successful open.

Mmap

The memory map routine in a device driver is responsible for returning a single physical page number of a portion of a device.

```c
/*ARGSUSED*/
int cgonemmap(dev, off, prot)
    dev_t dev;
    off_t off;
    int prot;
    {
        return(fbmmmap(dev, off, prot, NCGONE, cgoneinfo, CGLSIZE));
    }
```

cgonemmap used the utility fbmmmap.

```c
int fbmmmap(dev, off, prot, numdevs, mb_devs, size)
    dev_t dev;
    off_t off;
    int prot, numdevs, size;
    struct mb_device **mb_devs;
```

The parameters to fbmmmap are similar to fbopen. However, off is the offset in bytes from the beginning of the device. prot is passed through but currently not used.

ioctl

A pixrect kernel device driver must respond to two input/output control requests:

- FBIOGTYPE — Describe the characteristics of the pixel addressable device.
- FBIOGPIXRECT — Hand out a pixrect that may be used in the kernel. This ioctl call is made from within the kernel. This is only required of frame buffers.

```c
#define CGL_OPS &cgl_ops
struct pixrectops cgl_ops = {
    cgl_rop,        /*cgone.10*/
    cgl_putcolormap,
```
};
#endif

struct cglpr cgoneprdatadefault =
{ 0, 0, 255, 0, 0 };
struct pixrect cgonepixrectdefault =
{ CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };

struct pixrect cgonepixrect[NCGONE]; /*cgone.11*/
struct cglpr cgoneprdata[NCGONE];

cgoneioctl(dev, cmd, data, flag)
{ dev_t dev;
  caddr_t data;
  register int unit = minor(dev);
  switch (cmd) {
    case FBIOGTYPE:
      register struct fbtype *fb = (struct fbtype *)data;
      fb->fb_type = FBTYPE_SUNICOLOR;
      fb->fb_height = CG1_HEIGHT;
      fb->fb_width = CG1_WIDTH;
      fb->fb_depth = 8;
      fb->fb_cmsize = 256;
      fb->fb_size = CG1_HEIGHT*CG1_WIDTH;
      break;
    case FBIOGPIXRECT:
      register struct fbpixrect *fbpr
        = (struct fbpixrect *)data;
      register struct cglfb *cglfb =
        (struct cglfb *)cgoneinfo[(unit)]->md_addr;
      fbpr->fbpr_pixrect = &cgonepixrect[unit];/*cgone.12*/
      cgonepixrect[unit] = cgonepixrectdefault;/*cgone.13*/
      fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];/*cgone.14*/
      cgoneprdata[unit] = cgoneprdatadefault;/*cgone.15*/
      cgoneprdata[unit].cgpr_va = cglfb;/*cgone.16*/
      cgl_setreg(cglfb, CG_FUNCREG, CG_VIDEOENABLE);/*cgone.17*/
      cgl_intclear(cglfb); /*cgone.18*/
      break;
    default:
      return (ENOTTY);
  }
  return (0);
}
The SunView driver isn't configured into the system when NWIN = 0 (line cgone.9). When there is no SunView driver, don't reference the pixrect operations cgl_rop and cgl_putcolormap. The kernel version of cgl_rop (line cgone.10) only needs to be able to read and write memory pixrects for cursor management. Thus, you can

```c
#ifndef KERNEL
/* code not associated with reading and writing memory pixrects */
#endif KERNEL
```

to reduce the size of the code.

Memory for pixrect public (pixrect structure) and private (cglpr structure) objects is provided by arrays of each (line cgone.11) NCGONE long. A device n in these correspond to device n in cgoneinfo.

Lines cgone.12 through cgone.16 initialize a pixrect for a particular device. This ioctl call should enable video for a frame buffer (line cgone.17) and disable interrupts as well (line cgone.18).

Close

When the device is no longer being referenced, cgoneclose is called. All that is done is that the pixrect data structures of the device are zeroed.

```c
cgoneclose(dev, flag)
   dev_t dev;
   {
      register int unit = minor(dev);

      if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
         bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cglpr));
         bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
      }
   }
#endif
```

Plugging Your Driver into UNIX

You need to add the device driver procedures to cdevsw in /sys/sun/conf.c after assigning a new major device number to your driver:
#include "cgone.h"
#if NCGONE > 0
int cgoneopen(), cgonemmap(), cgoneioctl();
int cgoneclose();
#else
#define cgoneopen nodev
#define cgonemmap nodev
#define cgoneioctl nodev
#define cgoneclose nodev
#endif
{
    cgoneopen, cgoneclose, nodev, nodev, /*14*/
    cgoneioctl, nodev, nodev, 0,
    seltrue, cgonemmap,
},

Also, you need to add the new files associated with your driver to
/sys/conf/files.sun:

pixrect/cg1 colormap.c optional cgone win device-driver
pixrect/cg1 rop.c optional cgone win device-driver
sundev/cgone.c optional cgone device-driver

A.7. 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.

pr_clip modifies src->pos, dst->pos and dst->size so that all refer­
ences are to valid bits.

pr_clip(dstp, srcp)
    struct pr_subregion *dst;
    struct pr_prpos *src;

src->pr may be NULL.

Two operations on operations, reversesrc and reversedst, are provided
for adjusting the operation code to take into account video reversing of mono­
chrome pixrects of either the source or the destination.

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

These are implemented by table lookup 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.8. 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 memory, then we will go indirect by allocating a memory temporary, and then asking the destination to operate from there into itself.

- Case 2 -- writing to your frame buffer. This consists of 4 different cases depending on where the data is coming from: from nothing, 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.9. 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 chop off in left-x direction.

A.10. 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.11. Colormap

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

Monochrome

For monochrome raster devices, when pr_putcolormap is called, the convention is that if red[0] is zero then the display is light on dark, otherwise dark on light. For monochrome raster devices, when pr_getcolormap is called, the convention 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 color map. Otherwise, if the display is dark on light, then zero and -1 are reversed.

A.12. Attributes

pr_getattributes and pr_putattributes operations get/set a bitplane mask in color pixrects.

Monochrome

Monochrome devices ignore pr_putattribute calls that are setting the bitplane mask. Monochrome devices always return 1 when pr_getattribute asking for the bitplane mask.
A.13. Pixel  
pwo_get and pwo_put operations get/set a single pixel.

A.14. Stencil  
In its most efficient implementation, stencil code parallels rop code, all the while considering the 2 dimensional stencil. One way to implement stencil is to use rops. We pay a small efficiency penalty for this. You may not consider writing the special purpose code worthwhile for the bitmap stencils since they probably won’t get used nearly as much as rop. Here’s the basic idea (Temp is a temporary memory pixrect):

\[
\begin{align*}
\text{Temp} & = \text{Dest} \\
\text{Temp} & = \text{Dest} \oplus \text{Source} \\
\text{Temp} & = \text{Temp} \& \text{Stencil} \\
\text{Dest} & = \text{Dest} \& \neg \text{Stencil} \\
\text{Dest} & = \text{Dest} \mid \text{Temp}
\end{align*}
\]

i.e., \( \text{Dest} = (\text{Dest} \& \neg \text{Stencil}) \mid ((\text{Dest} \oplus \text{Source}) \& \text{Stencil}) \)

A.15. Curve  
pr_curve allows for generalized shape drawing.

A.16. Polygon  
pr_polyline is a natural extension to pr_vector. It is especially useful for devices that can optimize this operation.
Pixrect Functions and Macros
## Pixrect Functions and Macros

Table B-1  *Pixrect Library Functions and Macros - Part I*

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Bounding Box of Text String</td>
<td>Compute the bounding box of a text string</td>
<td><code>pf_textbound(bound, len, font, text)</code></td>
</tr>
<tr>
<td>Compute Location of Characters in Text String</td>
<td>Compute the location of characters in a text string</td>
<td><code>pf_textbatch(where, lengthp, font, text)</code></td>
</tr>
<tr>
<td>Compute Width and Height of Text String</td>
<td>Compute the width and height of a text string</td>
<td><code>pf_textwidth(len, font, text)</code></td>
</tr>
<tr>
<td>Create Memory Pixrect from an Image</td>
<td>Create a memory pixrect from an image</td>
<td><code>mem_point(width, height, depth, data)</code></td>
</tr>
<tr>
<td>Create Memory Pixrect</td>
<td>Create a memory pixrect</td>
<td><code>mem_create(w, h, depth)</code></td>
</tr>
<tr>
<td>Create Pixrect</td>
<td>Create a pixrect</td>
<td><code>pr_open(devicename)</code></td>
</tr>
<tr>
<td>Create Secondary Pixrect</td>
<td>Create a secondary pixrect</td>
<td><code>pr_region(pr, x, y, w, h)</code></td>
</tr>
<tr>
<td>Create Static Memory Pixrect</td>
<td>Create a static memory pixrect</td>
<td><code>mpr_static(name, w, h, depth, image)</code></td>
</tr>
</tbody>
</table>
Table B-1  Pixrect Library Functions and Macros — Part I—Continued

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw Textured Polygon</td>
<td><code>pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixrect *dpr, *spr;</td>
</tr>
<tr>
<td></td>
<td>int dx, dy</td>
</tr>
<tr>
<td></td>
<td>int nbnds, npts[];</td>
</tr>
<tr>
<td></td>
<td>struct pr_pos *vlist;</td>
</tr>
<tr>
<td></td>
<td>int op, sx, sy;</td>
</tr>
<tr>
<td>Draw Vector</td>
<td><code>pr_vector(pr, x0, y0, x1, y1, op, value)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int x0, y0, x1, y1, op, value;</td>
</tr>
<tr>
<td>Exchange Foreground</td>
<td><code>pr_reversevideo(pr, min, max)</code></td>
</tr>
<tr>
<td>and Background Colors</td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int min, max;</td>
</tr>
<tr>
<td>Get Colormap Entries</td>
<td><code>pr_getcolormap(pr, index, count, red, green, blue)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int index, count;</td>
</tr>
<tr>
<td></td>
<td>unsigned char red[], green[], blue[]</td>
</tr>
<tr>
<td>Get Memory Pixrect</td>
<td><code>mpr_linebytes(width, depth)</code></td>
</tr>
<tr>
<td>Data Bytes per Line</td>
<td>((pr_product(width, depth)+15)&gt;&gt;3) &amp;~1)</td>
</tr>
<tr>
<td>Get Pixel Value</td>
<td><code>pr_get(pr, x, y)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int x, y;</td>
</tr>
<tr>
<td>Get Plane Mask</td>
<td><code>pr_getattributes(pr, planes)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int *planes;</td>
</tr>
<tr>
<td>Get Pointer to Memory</td>
<td><code>mpr_d(pr)</code></td>
</tr>
<tr>
<td>Pixrect Data</td>
<td>((struct mpr_data *) (pr)-&gt;pr_data)</td>
</tr>
<tr>
<td>Initialize Raster File</td>
<td><code>pr_dump_init(input_pr, rh, colormap, type, copy_flag)</code></td>
</tr>
<tr>
<td>Header</td>
<td>struct pixrect *input_pr;</td>
</tr>
<tr>
<td></td>
<td>struct rasterfile *rh;</td>
</tr>
<tr>
<td></td>
<td>colormap_t *colormap;</td>
</tr>
<tr>
<td></td>
<td>int type, copy_flag;</td>
</tr>
<tr>
<td>Load Font</td>
<td><code>pf_open(name)</code></td>
</tr>
<tr>
<td></td>
<td>struct pixfont *pf_open(name)</td>
</tr>
<tr>
<td></td>
<td>char *name;</td>
</tr>
<tr>
<td>Load Private Copy of</td>
<td><code>pf_open_private(name)</code></td>
</tr>
<tr>
<td>Font</td>
<td>struct pixfont *pf_open_private(name)</td>
</tr>
<tr>
<td></td>
<td>char *name;</td>
</tr>
<tr>
<td>Load System Default</td>
<td><code>pf_default()</code></td>
</tr>
<tr>
<td>Font</td>
<td>struct pixfont *pf_default()</td>
</tr>
</tbody>
</table>

---

Version A of 17 February 1986
### Table B-1  Pixrect Library Functions and Macros – Part I—Continued

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Masked RasterOp</strong></td>
<td><code>#define pr_stencil(dpr, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy)</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pixrect *dpr, *stpr, *spr;</code></td>
</tr>
<tr>
<td></td>
<td><code>int dx, dy, dw, dh, op, stx, sty, sx, sy;</code></td>
</tr>
<tr>
<td><strong>Multiple RasterOp</strong></td>
<td><code>#define pr_batchrop(dpr, dx, dy, op, items, n)</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pixrect *dpr;</code></td>
</tr>
<tr>
<td></td>
<td><code>int dx, dy, op, n;</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pr-prpos items[];</code></td>
</tr>
<tr>
<td><strong>RasterOp</strong></td>
<td><code>#define pr_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pixrect *dpr, *spr;</code></td>
</tr>
<tr>
<td></td>
<td><code>int dx, dy, dw, dh, op, spr, sx, sy;</code></td>
</tr>
<tr>
<td><strong>Read Colormap from Raster File</strong></td>
<td><code>int pr_load_colormap(input, rh, colormap)</code></td>
</tr>
<tr>
<td></td>
<td><code>FILE *input;</code></td>
</tr>
<tr>
<td></td>
<td><code>struct rasterfile *rh;</code></td>
</tr>
<tr>
<td></td>
<td><code>colormap_t *colormap;</code></td>
</tr>
<tr>
<td><strong>Read Header from Raster File</strong></td>
<td><code>int pr_load_header(input, rh)</code></td>
</tr>
<tr>
<td></td>
<td><code>FILE *input;</code></td>
</tr>
<tr>
<td></td>
<td><code>struct rasterfile *rh;</code></td>
</tr>
<tr>
<td><strong>Read Image from Raster File</strong></td>
<td><code>struct pixrect *pr_load_image(input, rh, colormap)</code></td>
</tr>
<tr>
<td></td>
<td><code>FILE *input;</code></td>
</tr>
<tr>
<td></td>
<td><code>struct rasterfile *rh;</code></td>
</tr>
<tr>
<td></td>
<td><code>colormap_t *colormap;</code></td>
</tr>
<tr>
<td><strong>Read Raster File</strong></td>
<td><code>struct pixrect *pr_load(input, colormap)</code></td>
</tr>
<tr>
<td></td>
<td><code>FILE *input;</code></td>
</tr>
<tr>
<td></td>
<td><code>colormap_t *colormap;</code></td>
</tr>
</tbody>
</table>

### Table B-2  Pixrect Library Functions and Macros – Part II

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read Standard Raster File</strong></td>
<td><code>struct pixrect *pr_load_std_image(input, rh, colormap)</code></td>
</tr>
<tr>
<td></td>
<td><code>FILE *input;</code></td>
</tr>
<tr>
<td></td>
<td><code>struct rasterfile *rh;</code></td>
</tr>
<tr>
<td></td>
<td><code>colormap_t colormap;</code></td>
</tr>
<tr>
<td><strong>Release PixFont Resources</strong></td>
<td><code>pf_close(pf)</code></td>
</tr>
<tr>
<td><strong>Release Pixrect Resources</strong></td>
<td><code>struct pixfont *pf;</code></td>
</tr>
<tr>
<td><strong>Release Pixrect Resources</strong></td>
<td><code>#define pr_close(pr)</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pixrect *pr;</code></td>
</tr>
<tr>
<td><strong>Release Pixrect Resources</strong></td>
<td><code>#define pr_destroy(pr)</code></td>
</tr>
<tr>
<td></td>
<td><code>struct pixrect *pr;</code></td>
</tr>
<tr>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Replicated Source</td>
<td>pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)</td>
</tr>
<tr>
<td>RasterOp</td>
<td>struct pixrect *dpr, *spr; int dx, dy, dw, dh, op, sx, sy;</td>
</tr>
<tr>
<td>Set Background and Foreground Colors</td>
<td>pr_blackonwhite(pr, min, max)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int min, max;</td>
</tr>
<tr>
<td>Set Colormap Entries</td>
<td>define pr_putcolormap(pr, index, count, red, green, blue)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int index, count;</td>
</tr>
<tr>
<td></td>
<td>unsigned char red[], green[], blue[]</td>
</tr>
<tr>
<td>Set Foreground and Background Colors</td>
<td>pr_whiteonblack(pr, min, max)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int min, max;</td>
</tr>
<tr>
<td>Set Pixel Value</td>
<td>define pr_put(pr, x, y, value)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int x, y, value;</td>
</tr>
<tr>
<td>Set Plane Mask</td>
<td>define pr_putattributes(pr, planes)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int *planes;</td>
</tr>
<tr>
<td>Subregion Create Secondary Picrect</td>
<td>define struct pixrect *prs_region(subreg)</td>
</tr>
<tr>
<td></td>
<td>struct pr_subregion subreg;</td>
</tr>
<tr>
<td>Subregion Draw Vector</td>
<td>define prs_vector(pr, pos0, pos1, op, value)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int op, value;</td>
</tr>
<tr>
<td>Subregion Get Colormap Entries</td>
<td>define prs_getcolormap(pr, index, count, red, green, blue)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int index, count;</td>
</tr>
<tr>
<td></td>
<td>unsigned char red[], green[], blue[]</td>
</tr>
<tr>
<td>Subregion Get Pixel Value</td>
<td>define prs_get(srcprpos)</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos srcprpos;</td>
</tr>
<tr>
<td>Subregion Get Plane Mask</td>
<td>define prs_getattributes(pr, planes)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr; int *planes;</td>
</tr>
<tr>
<td>Subregion Masked RasterOp</td>
<td>define prs_stencil(dstregion, op, stenprpos, srcprpos)</td>
</tr>
<tr>
<td></td>
<td>struct pr_subregion dstregion;</td>
</tr>
<tr>
<td></td>
<td>int op;</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos stenprpos, srcprpos;</td>
</tr>
<tr>
<td>Name</td>
<td>Function</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Subregion Multiple RasterOp</strong></td>
<td>#define prs_batchrop(dstpos, op, items, n)</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos dstpos;</td>
</tr>
<tr>
<td></td>
<td>int op, n;</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos items[];</td>
</tr>
<tr>
<td><strong>Subregion RasterOp</strong></td>
<td>#define prs_rop(dstregion, op, srcprpos)</td>
</tr>
<tr>
<td></td>
<td>struct pr_subregion dstregion;</td>
</tr>
<tr>
<td></td>
<td>int op;</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos srcprpos;</td>
</tr>
<tr>
<td><strong>Subregion Release</strong></td>
<td>#define prs_destroy(pr)</td>
</tr>
<tr>
<td><strong>Pixrect Resources</strong></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td><strong>Subregion Replicated</strong></td>
<td>#define prs_replrop(dsubreg, op, sprpos)</td>
</tr>
<tr>
<td><strong>Source RasterOp</strong></td>
<td>struct pr_subregion dsubreg;</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos sprpos;</td>
</tr>
<tr>
<td><strong>Subregion Set Colormap Entries</strong></td>
<td>#define prs_putcolormap(pr, index, count, red, green, blue)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int index, count;</td>
</tr>
<tr>
<td></td>
<td>unsigned char red[], green[], blue[];</td>
</tr>
<tr>
<td><strong>Subregion Set Pixel Value</strong></td>
<td>#define prs_put(dstprpos, value)</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos dstprpos;</td>
</tr>
<tr>
<td></td>
<td>int value;</td>
</tr>
<tr>
<td><strong>Subregion Set Plane Mask</strong></td>
<td>#define prs_putattributes(pr, planes)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>int *planes;</td>
</tr>
<tr>
<td><strong>Trapezon RasterOp</strong></td>
<td>pr_traprop(dpr, dx, dy, t, op, spr, sx, sy)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *dpr, *spr;</td>
</tr>
<tr>
<td></td>
<td>struct pr_trap t;</td>
</tr>
<tr>
<td></td>
<td>int dx, dy, sx, sy, op;</td>
</tr>
<tr>
<td><strong>Write Header to Raster File</strong></td>
<td>int pr_dump_header(output, rh, colormap)</td>
</tr>
<tr>
<td></td>
<td>FILE *output;</td>
</tr>
<tr>
<td></td>
<td>struct rasterfile *rh;</td>
</tr>
<tr>
<td></td>
<td>colormap_t *colormap;</td>
</tr>
<tr>
<td><strong>Write Image Data to Raster File</strong></td>
<td>int pr_dump_image(pr, output, rh)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>FILE *output;</td>
</tr>
<tr>
<td></td>
<td>struct rasterfile *rh;</td>
</tr>
<tr>
<td><strong>Write Raster File</strong></td>
<td>int pr_dump(input_pr, output, colormap, type, copy_flag)</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *input_pr;</td>
</tr>
<tr>
<td></td>
<td>FILE *output;</td>
</tr>
<tr>
<td></td>
<td>colormap_t *colormap;</td>
</tr>
<tr>
<td></td>
<td>int type, copy_flag;</td>
</tr>
</tbody>
</table>
Table B-2  *Pixrect Library Functions and Macros – Part II—Continued*

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write Text and</strong></td>
<td><strong>Background</strong></td>
</tr>
<tr>
<td></td>
<td>pf_text(where, op, font, text)</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos where;</td>
</tr>
<tr>
<td></td>
<td>int op;</td>
</tr>
<tr>
<td></td>
<td>struct pixfont *font;</td>
</tr>
<tr>
<td></td>
<td>char *text;</td>
</tr>
<tr>
<td><strong>Write Text</strong></td>
<td>pf_ttext(where, op, font, text)</td>
</tr>
<tr>
<td></td>
<td>struct pr_prpos where;</td>
</tr>
<tr>
<td></td>
<td>int op;</td>
</tr>
<tr>
<td></td>
<td>struct pixfont *font;</td>
</tr>
<tr>
<td></td>
<td>char *text;</td>
</tr>
</tbody>
</table>
Pixrect Data Structures
## Pixrect Data Structures

### Table C-1  Pixrect Data Structures

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| **Character Descriptor** | struct pixchar {
|                       |   struct pixrect *pc_pr;
|                       |   struct pr_pos pc_home;
|                       |   struct pr_pos pc_adv;
|                       | }; |
| **Font Descriptor**   | struct pixfont {
|                       |   struct pr_size pf_defaultsize;
|                       |   struct pixchar pf_char[256];
|                       | }; |
| **Pixrect**            | struct pixrect {
|                       |   struct pixrectops *pr_ops;
|                       |   struct pr_size pr_size;
|                       |   int pr_depth;
|                       |   caddr_t pr_data;
|                       | }; |
| **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)();
|                       |   int (*pro_put colormap)();
|                       |   int (*pro_get colormap)();
|                       |   int (*pro_put attributes)();
|                       |   int (*pro_get attributes)();
<p>|                       | }; |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>struct pr_pos { int x, y; }</td>
</tr>
<tr>
<td>Position Within a</td>
<td>struct pr_prpos {</td>
</tr>
<tr>
<td>Pixrect</td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>struct pr_pos pos;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Size</td>
<td>struct pr_size { int x, y; }</td>
</tr>
<tr>
<td>Subregion</td>
<td>struct pr_subregion {</td>
</tr>
<tr>
<td></td>
<td>struct pixrect *pr;</td>
</tr>
<tr>
<td></td>
<td>struct pr_pos pos;</td>
</tr>
<tr>
<td></td>
<td>struct pr_size size;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Trapezon</td>
<td>struct pr_trap {</td>
</tr>
<tr>
<td></td>
<td>struct pr_fall *left, *right;</td>
</tr>
<tr>
<td></td>
<td>int y0, y1;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Trapezon Chain</td>
<td>struct pr_chain {</td>
</tr>
<tr>
<td></td>
<td>struct pr_chain *next;</td>
</tr>
<tr>
<td></td>
<td>struct pr_size size;</td>
</tr>
<tr>
<td></td>
<td>int *bits;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Trapezon Fall</td>
<td>struct pr_fall {</td>
</tr>
<tr>
<td></td>
<td>struct pr_pos pos;</td>
</tr>
<tr>
<td></td>
<td>struct pr_chain *chain;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Curved Shapes
This appendix\(^1\) describes \texttt{pr\_traprop}, a function for rendering curved shapes with \texttt{Pixrect}. \texttt{pr\_traprop} is an advanced \texttt{pixrect} operation analogous to \texttt{pr\_rop}.

The curve to be rendered must first be stored in a data structure called \texttt{pr\_trap} which is based on a region called a \textit{trapezon}, rather than on a rectangle. A trapezon is a region with an irregular boundary. Like a rectangle, a trapezon has four sides: top, bottom, left, and right. The top and bottom sides of a trapezon are straight and horizontal. A trapezon differs from a rectangle in that its left and right sides are irregular curves, called \textit{falls}, rather than straight lines.

A fall is a line of irregular shape. Vertically, a fall may only move downward. Horizontally, a fall may move to the left or to the right, and this horizontal motion may reverse itself. A fall may also sustain pure horizontal motion, that is, horizontal motion with no vertical motion.

The figures below show a typical trapezon with source and destination \texttt{pixrects}, and some examples of filled regions that were drawn by \texttt{pr\_traprop}.

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node[anchor=north west,inner sep=0] (image) at (0,0) {
\includegraphics[width=\textwidth]{example-image-a}
};
\begin{scope}[x={(image.south east)},y={(image.north west)}]
\node at (0.5,0.8) {\texttt{source pr}};
\node at (0.5,0.1) {\texttt{dest pr}};
\node at (0.5,0.6) {dx,dy};
\node at (0.5,0.3) {op};
\node at (0.6,0.7) {sx,sy};
\end{scope}
\end{tikzpicture}
\caption{Typical Trapezon}
\end{figure}

\(^1\) The functionality of curve support in \texttt{Pixrect} may change in the future.
Some Figures Drawn by \texttt{pr\_traprop}

\begin{figure}[h]
\centering
\begin{tabular}{cccc}
\includegraphics[width=0.2\textwidth]{fig1.png} & \includegraphics[width=0.2\textwidth]{fig2.png} & \includegraphics[width=0.2\textwidth]{fig3.png} & \includegraphics[width=0.2\textwidth]{fig4.png} & \includegraphics[width=0.2\textwidth]{fig5.png} \\
\end{tabular}
\caption{Some Figures Drawn by \texttt{pr\_traprop}}
\end{figure}

\begin{verbatim}
\texttt{pr\_traprop}(dpr, dx, dy, t, op, spr, sx, sy)
struct pixrect \*dpr, \*spr;
struct pr\_trap t;
int dx, dy, sx, sy op;

\texttt{dpr} and \texttt{spr} are pointers to the destination and source pixrects, respectively. \texttt{t} is the trapezon to be used. \texttt{dx} and \texttt{dy} specify an offset into the destination pixrect. \texttt{sx} and \texttt{sy} specify an offset into the source pixrect. \texttt{op} is an op-code as specified previously (see Section 2.6).

struct pr\_trap {
    struct pr\_fall \*left, \*right;
    int y0, yl;
};

struct pr\_fall {
    struct pr\_pos pos;
    struct pr\_chain \*chain;
};

struct pr\_chain {
    struct pr\_chain \*next;
    struct pr\_size size;
    int \*bits;
};

\texttt{pr\_traprop} performs a rasterop from the source to the destination, clipped to the trapezon's boundaries. A program must call \texttt{pr\_traprop} once per trapezon; therefore this procedure must be called at least twice to draw the letter 'A' in Figure D-2.

The source pixrect is aligned with the destination pixrect; the pixel at \((sx, sy)\) in the source pixrect goes to the pixel at \((dx, dy)\) in the destination pixrect (see Figure D-2).

Positions within the trapezon are relative to position \((dx, dy)\) in the destination pixrect. Thus, a position defined as \((0,0)\) in the trapezon would actually be at \((dx, dy)\) in the destination pixrect.
\end{verbatim}
The structure `pr_trap` defines the boundaries of a trapezon. A trapezon consists of pointers to two falls (`left` and `right`) and two y coordinates specifying the top and bottom of the trapezon (`y0` and `y1`). Note that the trapezon's top and bottom may be of zero width; `y0` and `y1` may simply serve as points of reference.

Each fall consists of a starting position (`pos`) and a pointer to the head of the list of chains describing the path the fall is to take (`chain`). A fall may start anywhere above the trapezon and end anywhere below it. `pr_traprop` ignores the portions of a fall that lie above and below the trapezon. If a fall is shorter than the trapezon, `pr_traprop` will clip the trapezon horizontally to the endpoint of the fall in question. Figure D-3 illustrates the way this works.

A chain is a member of a linked list of structures that describes the movement of the fall. Each chain describes a single segment of the fall. Each chain consists of a pointer to the next member of the chain (`next`), the size of the bounding box for the chain (`size`), and a pointer to a bit vector containing motion commands (`bits`). See Section 1.3 for a description of the `pr_size` structure.

Each chain may specify motion to the right and/or down, or motion to the left and/or down; however, a single chain may not specify both rightward and leftward motion. Remember that motion may not proceed upward, and that straight horizontal motion is permitted.

The x value of the chain’s `size` determines the direction of the motion: a positive x value indicates rightward motion, while a negative x value indicates leftward motion. The y value of the chain’s `size` must always be positive, since a fall may not move upward (in the direction of negative y).

A chain’s bit vector is a command string that tells `pr_traprop` how to draw each segment of the fall. Each set (1) bit in the vector is a command to move one pixel horizontally and each clear (0) bit is a command to move one pixel vertically. The bits within the bit vector are stored in byte order, from most significant bit to least significant bit. This ordering corresponds to the left-to-right ordering of pixels within a memory `pixrect`.

The fall begins at the starting position specified in `pr_fall`. The motion proceeds downward as specified in the first bit vector in the chain, from the high-order bit to the low-order bit. When the fall reaches the bottom of the bounding box, it continues at the top of the next chain’s bounding box. Note that the fall will always begin and end at diagonally opposite corners of a given bounding box.

If a bit vector specifies a segment of the fall that would run outside of the bounding box, `pr_traprop` clips that segment of the fall to the bounding box. This would occur when the sum of the 1’s in a chain’s bit vector exceeds the chain’s x size, or when the sum of the 0’s in the chain’s bit vector exceeds the chain’s y size. When this happens, the segment in question runs along the edge of the bounding box until it reaches the corner of the bounding box diagonally opposite to the corner in which it started.

If the fall has a straight vertical segment, the x size of its chain must be 0. If the fall has a straight horizontal segment, the y size of its chain must be 0.
Figure D-3  *Trapezon with Clipped Falls*

The following program draws the octagon in the middle of the display surface.
Example Program with pr_traprop

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

int shallowsteep[] = {0bbbbbbbb, 0bbbbbbbb, 
                      04444444, 04444444};
int steepshallow[] = {04444444, 04444444, 
                      0bbbbbbbb, 0bbbbbbbb};

struct pr_chain leftl = {0, {64, 64}, steepshallow}, 
                        left0 = {&leftl, {-64, 64}, shallowsteep}, 
                        rightl = {0, {-64, 64}, steepshallow}, 
                        right0 = {&rightl, {64, 64}, shallowsteep};

struct pr_fall left_oct = {{0, 0}, &left0}, 
                          right_oct = {{0, 0}, &right0};

struct pr_trap octagon = {&left_oct, &right_oct, 0, 128};

main()
{
    struct pixrect *screen;

    screen = pr_open("/dev/fb");
    pr_traprop(screen, 576, 450, octagon, PIX_SET, 0, 0, 0);
    pr_close(screen);
}
```

pr_chain specifies the left lower, the left upper, the right lower, and the right upper sides of the octagon, in that order. pr_fall specifies first the left side, then the right side of the octagon.

Each of the eight sides of the octagon is half a chain. The two upper left sides correspond to chain left0. The bits start out with mostly 1’s (0xb is binary 1011) for the shallow uppermost left edge. They turn to mostly 0’s (0x4 is binary 0100) for the next edge down, which is steeper.
Index

Special Characters
<pixrect/memvar.h>, 36
<pixrect/pixrect.h>, 5
<rasterfile.h>, 41
<stdio.h>, 41

B
background, 22
bitplane, 23
bitplane mask, 23

C
compiling Pixrect, 5
Compute Bounding Box of Text String, 32, 73
Compute Location of Characters in Text String, 31, 73
Compute Width and Height of Text String, 32, 73
Create Memory Pixrect, 36, 73
Create Memory Pixrect from an Image, 36, 73
Create Pixrect, 11, 73
Create Secondary Pixrect, 11, 73
Create Static Memory Pixrect, 37, 73
curved shapes, 85

D
documentation conventions, xi
Draw Textured Polygon, 19, 73
Draw Vector, 18, 73

E
Exchange Foreground and Background Colors, 23, 73

F
fbintr, 64
fbbmap, 65
fbopen, 65
font, 18, 29, 31
fontedit, 30
foreground, 22

G
Get Colormap Entries, 22, 73
Get Memory Pixrect Data Bytes per Line, 36, 73
Get Pixel Value, 12, 73
Get Plane Mask, 24, 73
Get Pointer to Memory Pixrect Data, 36, 73

I
Initialize Raster File Header, 45, 73

L
lint library, 5
Load Font, 30, 73
Load Private Copy of Font, 30, 73
Load System Default Font, 30, 73

M
Masked RasterOp, 16, 73
mem_create, 36, 73
mem_point, 36, 73
memory pixrects, 35, 36
mpr_d, 36, 73
mpr_data, 35
mpr_linebytes, 36, 73
mpr_static, 37, 73
Multiple RasterOp, 17, 73

P
pf_close, 31, 75
pf_default, 30, 73
pf_open, 30, 73
pf_open_private, 30, 73
pf_text, 31, 75
pf_textbatch, 31, 73
pf_textbound, 32, 73
pf_textwidth, 32, 73
pf_ttext, 31, 75
PIX_CLR, 13
PIX_COLOR, 14
PIX_DONTCLIP, 14
PIX_DST, 13
PIX_NOT, 13
PIX_OPCOLOR, 14
PIX_SET, 13
PIX_SRC, 13
pixchar, 29, 81
pixfont, 29, 81
pixrect, 81
Pixrect
Pixrect, continued
audience, xi
compiling, 5
header file, 5
lint library, 5
writing a device driver, 51

pixrectops, 9, 10, 81
pr_batchrop, 17, 73
pr_blackonwhite, 23, 75
pr_chain, 81, 86
pr_clip, 68
pr_close, 12, 75
pr_destroy, 12, 75
pr_dump, 41, 75
pr_dump_header, 45, 75
pr_dump_image, 46, 75
pr_dump_init, 45, 73
pr_fail, 81, 86
pr_get, 12, 73
pr_getattributes, 24, 73
pr_getcolormap, 22, 73
pr_load, 43, 73
pr_load_colormap, 46, 73
pr_load_header, 46, 73
pr_load_image, 46, 73
pr_load_std_image, 47, 75
pr_makefromfd, 56
pr_open, 11, 73
pr_polygon_2, 19, 73
pr_pos, 81
pr_ppos, 81
pr_put, 12, 75
pr_putattributes, 24, 75
pr_putcolormap, 22, 75
pr_region, 11, 73
pr_replrop, 16, 75
pr_reversesdst, 68
pr_reversesrc, 68
pr_reversevideo, 23, 73
pr_rop, 15, 73
pr_size, 81
pr_stencil, 16, 73
pr_subregion, 81
pr_trap, 81, 86
pr_traprop, 75, 85
pr_unmakefromfd
  pr_unmakefromfd, 58
pr_vector, 18, 73
pr_whiteonblack, 23, 75
primary pixrect, 11

prs_batchrop, 17, 75
prs_destroy, 12, 75
prs_get, 12, 75
prs_getattributes, 24, 75
prs_getcolormap, 22, 75
prs_put, 12, 75
prs_putattributes, 24, 75
prs_putcolormap, 22, 75
prs_region, 11, 75
prs_replrop, 16, 75
prs_rop, 15, 75
prs_stencil, 16, 75
prs_vector, 18, 75

R
rasterfile, 44
RasterOp, 15, 73
Read Colormap from Raster File, 46, 73
Read Header from Raster File, 46, 73
Read Image from Raster File, 46, 73
Read Raster File, 43, 73
Read Standard Raster File, 47, 75
Release Psfont Resources, 31, 75
Release Pixrect Resources, 12, 75
Replicated Source RasterOp, 16, 75

S
secondary pixrect, 11
Set Background and Foreground Colors, 23, 75
Set Colormap Entries, 22, 75
Set Foreground and Background Colors, 23, 75
Set Pixel Value, 12, 75
Set Plane Mask, 24, 75
Subregion Create Secondary Pixrect, 11, 75
Subregion Draw Vector, 18, 75
Subregion Get Colormap Entries, 22, 75
Subregion Get Pixel Value, 12, 75
Subregion Get Plane Mask, 24, 75
Subregion Masked RasterOp, 16, 75
Subregion Multiple RasterOp, 17, 75
Subregion RasterOp, 15, 75
Subregion Release Pixrect Resources, 12, 75
Subregion Replicated Source RasterOp, 16, 75
Subregion Set Colormap Entries, 22, 75
Subregion Set Pixel Value, 12, 75
Subregion Set Plane Mask, 24, 75

T
trapezon, 85
Trapezon RasterOp, 75, 85

W
Write Header to Raster File, 45, 75
Write Image Data to Raster File, 46, 75
Write Raster File, 41, 75
Write Text, 31, 75
Write Text and Background, 31, 75
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2/17/86</td>
<td>3.0 Production Release.</td>
</tr>
</tbody>
</table>
Notes
Notes
Notes
Notes