sun® microsystems

Documentation Errata and Change Pages for SunOS Release 4.0.3

Part No: 800-3377-10
Revision A of 24 April 1989

Documentation Errata and Change Pages for SunOS Release 4.0.3

Introduction

The ''mini-box'' of 4.0.3 release documentation contains this document and two manuals, *Installing the SunOS 4.0.3* and the *SunOS 4.0.3 Release Manual.* The manuals contain information you will need in order to upgrade your system. This document, ''Documentation Errata and Change Pages for SunOS Release 4.0.3,'' supplements and corrects the documentation set for SunOS™ 4.0.3 in the following areas:

- **CG8 Release Notes**

  Describes the changes to Pixrect, SunView 1, and Sun diagnostic software to support the new `cg8` 24-bit frame buffer.

- **Sun-2 and Sun-3 Assembler**

  Describes changes to the *Assembly Language Reference Manual* to support the MC68030 microprocessor included in the `as` assembler for Sun-2 and Sun-3 workstations.

- **Device Drivers**

  Describes corrections required to parts of *Writing Device Drivers.*

Refer also to the *READ THIS FIRST* document, which is packed in the release-tape box. *READ THIS FIRST* contains information that, because of publication deadlines, could not be included elsewhere.

`cg8` 24-Bit Color

This section describes the changes implemented in Pixrect, SunView 1, and diagnostic software to support the new 24-bit frame buffer, the `cg8`. Refer to the *SunView 1 Programmer's Guide*, the *SunView 1 System Programmer's Guide*, and the *Pixrect Reference Manual* for more information about Pixrect and SunView 1 in SunOS 4.0.3. For more information about diagnostics, refer to the *Sun System Diagnostics* manual.

Overview

The `cg8` color board has three planes: a 24-bit plane to represent *true color* images, a 1-bit overlay plane for the high-speed display of monochrome images, and an enable plane for selecting between the other two planes.

Pixrect Support

The changes to the Pixrect package have been deliberately kept to a minimum. The `cg8` board stores its color pixels in a format called XBGR. Pixrect now understands this format, storing XBGR format pixels in 32-bit pixrects.

The true color lookup table is manipulated with two new macros. There is also a new Pixrect planegroup, called `PIXPG_24BIT_COLOR`, which provides access to the `cg8` true-color plane.

XBGR Format

The Pixrect library already supports 1-bit, 8-bit, and 32-bit-deep pixrects (32 as memory pixrects). Since true color pixrects are stored in a format that is 32 bits deep, there were minimal changes made to the Pixrect library. A new type of pixel format, called XBGR is defined to hold true color pixrects. The `cg8` stores 24-bit images in this format. The format is shown below:

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

union fbunit {
    u_int               packed;  normal 32-bit pixrects
    struct {
    u_int               A:8;     the high-order 8 bits are unused
    u_int               B:8;     8 bits of blue component
    u_int               G:8;     8 bits of green component
    u_int               R:8;     8 bits of red component
    }                   channel;
};
```

The 32-bit word is divided into 4 *channels* of 8 bits each. The first channel (the high-order 8 bits) is not currently used by the `cg8`. Its value is undefined, and it is reserved for future enhancements. The next channel contains 8 bits of the pixel blue component (256 possible values, from 0 to 255, for the blue component of the pixel color). The other two channels hold corresponding information for the green and red components of the pixel color. The three components are used to index the red green and blue parts of the lookup table. The RGB components from the lookup table are combined to produce a pixel with a particular hue and intensity.

Figure 1 *XBGR Layout*

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

The op Argument

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

Source Depth		*Destination Depth*	*Operations Allowed*
0	→	*n*	yes
1	→	*n*	yes
n	→	1	not allowed
n	→	*n*	yes
24	→	32	not allowed
32	→	24	not allowed

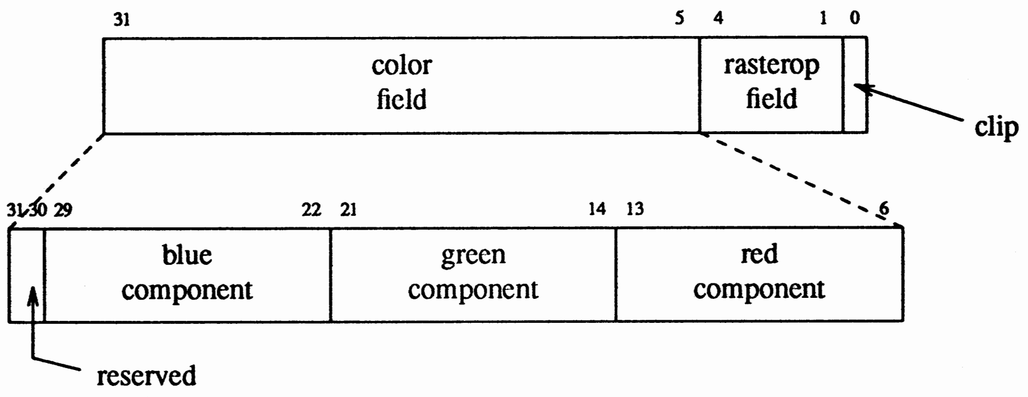
The value *n* can be the values 1, 8, and 32 bits. Note that 8 to 32 and 32 to 8 are **not** supported. To translate pixel colors between 8 and 32, you must use the formula shown below. It uses the 8-bit pixel value (the variable *color8*), in conjunction with the 8-bit colormap, to generate a 24-bit color (saved in the integer variable *color24*). The *color24* variable now has its true color stored in XBGR format. The value can then be saved as a 32-bit pixel, in the pixrect PIXPG_24BIT_COLOR planegroup.

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

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

The *color* field of the op argument, used in many pixrect functions, is interpreted differently on the cg8 frame buffer. The most significant 8 bits of this field (bit 24 to 31 of op) is the value of the blue channel, the next 8 bits (bit 16 to 23) is the value of the green channel and the next 8 bits (bit 8 to bit 15) is the value of the red channel.

Figure 2 op *fields*



The Pixrect library cannot distinguish between indexed 8-bit color and 24-bit direct color applications. Do not expect your current 8-bit color programs to use 24-bit color without modification.

24 vs 32 bits

Since true color pixrects use only 24 bits per pixel to encode color information, yet are stored and handled as 32-bit entities, a number of compatibility issues come up.

Memory Pixrects

It is possible to create 24-bit *memory* pixrects. This may be useful for synthesizing images that are later displayed. 24-bit-to-24-bit operations are supported; you cannot, however, perform a rop operation between a 24-bit pixrect and a 32-bit one, even if the 32-bit pixrect is in XBGR format.

Sometimes, it is more efficient to use a 24-bit memory pixrect to generate an image, then save it as a 24-bit raster file. When pr_load() is called to load a 24-bit raster file, however, it automatically loads it in as a 32-bit pixrect so that pixrect operations run more efficiently. When pr_save() is called, the converted pixrect will be saved in a 32-bit raster file.

Run-Length Encoding

It makes little sense to save a 32-bit pixrect (in XBGR format) as a run-length encoded raster file, although there is no technical difficulties doing so. Since it is a *true-color* pixrect, it is unlikely any adjacent *bytes* have the same value, even if the adjacent XBGR *pixels* do. The run-length encoding compression scheme is byte-oriented, not pixel-oriented, so it is not likely to reduce the size of the file.

Plane Groups

Pixrect has added a new plane group, `PIXPG_24BIT_COLOR`, for access to the `cg8` 24-bit buffer. The new plane group provides 24-bit RGB (red green blue) values stored in XBGR format in 32-bit pixels. All the normal logical operations and plane masking are available; many of the logical operations are not useful with color, however.

The `cg8` also has an overlay and an enable plane, providing three planegroups for `cg8` pixrects:

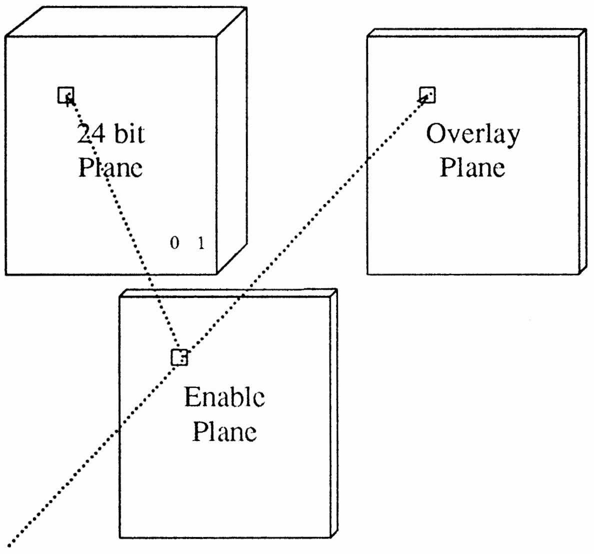
Table 1 `cg8` *planegroups*

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

The overlay plane is black and white by default. It does, however, have its own 2-bit colormap. The foreground and background colors of the overlay plane can be set using the `pr_putcolormap()` command with the overlay plane as the pixrect argument. The colormap contains two 24-bit color values; one for pixels set to 1, and one for pixels set to 0.

The enable and overlay planes have the same behavior seen in previous Sun frame buffers (`cg4`); the enable plane mediates between the overlay and the 24-bit plane. When an enable pixel is one, only the overlay plane is visible; when it is zero, only the 24-bit plane is visible.

Figure 3 `cg8` *planegroups*



NOTE: The default plane for the `cg8` is the overlay plane, **not** the 24-bit plane.

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

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

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

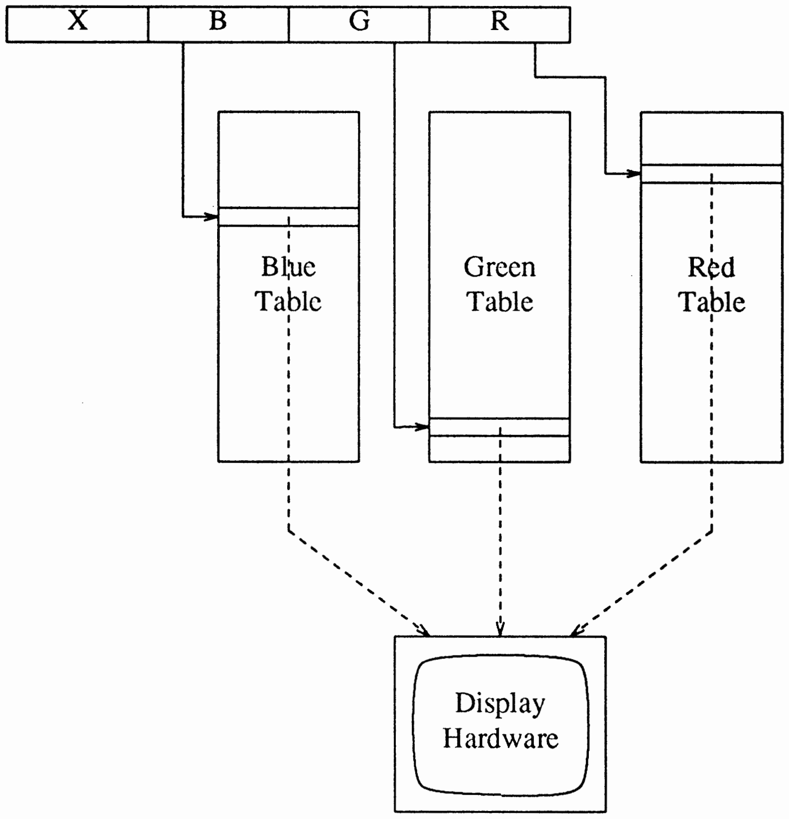
Lookup Tables

The `cg8` has three lookup tables that can be used to adjust the intensity response of each component (red, green, and blue) of the 24-bit color values. These tables can be used to make the color response nonlinear (for example in gamma correction), or for special color effects.

For each color component, the intensity value of the pixel is used as an index to the corresponding table entry. The value of that entry is then used as the actual intensity component for the displayed pixel. This can be used to compensate for color inaccuracies generated by the display hardware; thus by applying color correction techniques (like gamma correction) with the lookup table, the displayed image approaches true color.

Both 24-bit color *lookup tables* and 8-bit color *colormaps* require 768 bytes of space. The Pixrect functions `pr_getcolormap()` and `pr_putcolormap()` commands are used to read or modify 8-bit colormaps, while `pr_getlut()` and `pr_putlut()` are the corresponding commands to read and modify 24-bit lookup tables.

Figure 4 *True-Color Lookup*



Using the `pr_putlut()` macro to load the lookup tables is similar to using the `pr_putcolormap()` function; the ***red[]***, ***green[]***, and ***blue[]*** array arguments correspond to the appropriate lookup tables. In the same way, `pr_getlut()` loads these same arrays from the lookup tables.

Upon opening a pixrect, the `cg8` lookup table is loaded with linear ramps from 0 to 255 for the red, green, and blue tables. The default table therefore has no built-in corrections. All applications share the same lookup table; you cannot divide it up into portions as you can with 8-bit colormaps, and lookup table segmenting is not allowed in Pixrect or SunView1 running on a `cg8`.

The lookup table segment size is fixed at 256; no lookup segmenting is allowed.

Two pixrect lookup macros, `pr_putlut()` and `pr_getlut()` are shown below:

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

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

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

The PR_FORCE_UPDATE value in the pr_putlut() macro is necessary because there is no colormap sharing in Pixrect. The sample program below shows the macros in use:

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

pr = pr_open("/dev/cgeight0");
pr_set_plane_group(pr, PIXPG_24BIT_COLOR); change to 24-bit plane
pr_getlut(pr, 0, 256, red, green, blue);
gamma_correct(red,green,blue); a user-supplied function...
pr_putlut(pr, 0, 256, red, green, blue);
```

The code fragment above opens the cg8 frame buffer, and changes the current plane group to be 24-bit color (the default is the overlay plane). The pr_putlut() and pr_getlut() macros read, then reload the lookup tables.

Indexed 24-bit Color

At this time, indexed 24-bit color (the 24-bit equivalent to 8-bit color) is **not** supported in Pixrects or SunView1.

SunView1

The 24-bit SunView1 model is less general than the 8-bit one. SunView1 on the cg8 is a monochrome desktop residing in the overlay plane. The color canvases in the desktop are 24-bit color, all affected by the same lookup table.

Overlay Colormap

Although the SunView1 desktop is restricted to two colors (normally black and white) there *is* a 2 bit colormap for the overlay plane. This means the programmer can set the foreground and background colors of the desktop by using the pw_putcolormap() command with the overlay plane as the pixrect argument. The table contains two 24-bit color values; one for pixels set to 1, and one for pixels set to 0.

Modifying the Lookup Table

There is no lookup table sharing implemented in the 24-bit SunView1 system. Any changes in the lookup table affect all systems. SunView1 **cannot** modify the lookup table directly. The pw_putcolormap() command is ignored when accessing the lookup table. There is no pw_putlut() macro corresponding to the Pixrect pr_putlut() macro. To change the lookup table, the SunView1 application must make a pr_putlut() call with Pixrect. To do this, the application must extract the pixrect pointer from the pixwin data structure, then use that pointer in a pr_putlut() call. An example code fragment that performs gamma correction is shown below. We assume the user has written a function called gamma_correct() which adjusts the red, green and blue values of the lookup table.

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

pr_set_plane_group(pw->pw_pixrect, PIXPG_24BIT_COLOR);
if(pr_get_plane_group(pw->pw_pixrect) != PIXPG_24BIT_COLOR)
    return(0);
pr_getlut(pw->pw_pixrect, 0, 255, red, green, blue);
gamma_correct(red, green, blue);
pr_putlut(pw->pw_pixrect, 0, 255, red, green, blue);
```

This example uses the pixwin from the window canvas, extracts the pixrect, and switches to the 24-bit planegroup. If the planegroup is not `PIXPG_24BIT_COLOR`, then the display device does not support 24-bit color, and the rest of the program is aborted. If it does have true color, the code uses `pr_getlut()` with the extracted pixrect to get the contents of the lookup table, uses `gamma_correct()` to adjust the values, then calls `pr_putlut()`, to alter the lookup table.

See the Pixrect example code near the end of this section for more details in using `pr_putlut()` and `pr_getlut()`.

All `pw_putcolormap()` and `pw_getcolormap()` calls are ignored when the call is accessing a 24-bit plane-group. No error is reported. This allows older 8-bit color applications to fail ''gracefully'' when run on the `cg8`.

SunView1 Startup

When starting a SunView1 desktop on a `cg8` device, only 24-bit true color is allowed. Thus the following SunView1 flags are disabled:

Figure 5 *Disabled SunView1 Flags*

```
sunview   -8bit_color_only
          -overlay_only
          -toggle_enable
```

The 24-bit Canvas

The basic sunview window requirement is to set the `CANVAS_COLOR24` attribute to `TRUE`. This tells the window system that this will be a true color window. The window canvas will then reside in the `cg8` `PIXPG_24BIT_COLOR` plane group. The initial lookup table segment is `CMS_COLOR_IDENTITY`.

Using `CANVAS_COLOR24` runs the window canvas in the 24-bit frame, and using `CANVAS_FAST_MONO` puts the canvas in the overlay plane. If both attributes are set, the one most recently set takes effect.

All 24-bit canvases use the same lookup table; any change to this table will affect all canvases running on the `cg8`. The backing pixrect for 24-bit canvases will be 32 bits deep, storing the pixels in the `XBGR` format described earlier.

Any windows not designated as 24-bit canvases will be monochrome, using the colormap of the overlay plane.

Porting from 8-bit color to 24-bit color

Eight bit Color applications must be modified if they are to work properly on the `cg8`. Since 8-bit color uses a color-map, while 24-bit color uses a lookup table, the color values generated by the program should be translated using the following formula:

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

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

The *color8* variable is the 8 bit pixel value to be translated. The *red[]*, *green[]*, and *blue[]* arrays are the contents of the colormap used with the 8-bit pixels being translated. The *color24* variable holds the resulting 24-bit color in `XGBR` format. This value can be loaded into a 32-bit pixel, in the pixrect `PIXPG_24BIT_COLOR` planegroup.

Basically, this formula translates the 8-bit index into the red, green and blue components that would be displayed on the screen of an 8-bit color system. These translated color components are then loaded into the appropriate bit locations for a 24-bit color value in XGBR format.

Porting Limitations

While the translation approach is probably the fastest way make 8-bit color applications portable to 24 bits, it is not the best. First, you are not taking advantage of all the additional colors available on the `cg8`. More importantly, the 24-bit system is going to be slower by about a factor of four (moving 32-bit, instead of 8-bit, pixels). Adding an additional translation step will slow it down further.

It would be better to change the application model so it understands 24-bit color in the first place. It could be set to cut down its 24-bit colors to 8 when it runs on 8-bit color boards (for portability).

Diagnostics

The two sections below describe the diagnostics available for the `cg8` frame buffer.

Boot PROMs

The `cg8` is the first frame buffer to utilize an external boot PROM. The external boot PROM allows all frame buffer specific code to placed in its own the frame buffer PROM. This allows the host machine to test the frame buffer without knowing its configurations details.

The boot sequence has been changed to accommodate the new boot code. At boot up, host machine self-tests are run normally, but the frame buffer selftests are no longer part of the host's PROM.

Once the host machine performs its own self-test, the host CPU probes all external devices for external boot PROMs. When it finds a frame buffer, it loads the frame buffer code into its own memory. After the code is loaded, the host CPU initializes all frame buffer data, such as manufacturer, model number, revision, and resolution, then moves the information into an area accessible to the operating system. The host machine then initializes the display routines.

At this point, the frame buffer self-tests are executed. The table below shows the tests that are executed, along with their frame buffer status codes.

NOTE The DM notation indicates this test is only run if the diagnostic mode switch is enabled.

Table 2 *Frame buffer codes*

`08`	***Enable memory tests.***	
`08`	Quick address line test.	
`09`	Quick data line test.	
`0A`	Long address pattern test.	DM
`0B`	Data size test.	DM
`0C`	Data bits test.	DM
`0D-0F`	Reserved for future expansion.	
`1`	***Overlay memory tests.***	
`1`	Quick address line test.	
`1`	Quick data line test.	
`1`	Long address pattern test.	DM
`1`	Data size test.	DM
`1`	Data bits test.	DM
`15-1`	Reserved for future expansion.	
`1`	***Color memory tests.***	
`1`	Quick address line test.	
`1`	Quick data line test.	
`1A`	Long address pattern test.	DM
`1B`	Data size test.	DM
`1C`	Data bits test.	DM
`10-1F`	Reserved for future expansion	
`2`	***Color controller (ramdac test).***	
`2`	Color palette tests.	
`29-3F`	Reserved for future expansion.	
`40-FF`	Reserved for future expansion.	

The following is the output from the selftests while the diagnostic switch is in the `diag` position:

```
Sizing Memory (size = 0x00000008 Megabytes).

Selftest Completed.

Type a key within 10 secondsto enter Extended Test System (e for echo mode).
Looking for display devices
cgeight
Testing display
Testing cgeight
  Overlay-plane
    Data lines test
    Address quick test
    Data size test
    Data bits test
    Address=data test
  Enable-plane
    Data lines test
    Address quick test
    Data size test
    Data bits test
    Address=data test
```

Sysdiag

The `cg8` is tested in `sysdiag` by a test called `color24`. This test verifies all device-specific driver routines, colormaps, and frame buffer memories. Since all tests in `sysdiag` are executed automatically, no user intervention is required to run `color24`.

Pixrect Example Code

As the number of bits per pixel increases, the amount of data stored in a pixrect becomes significant. The size of a XGBR-format pixrect is substantial; thus, both space efficiency and performance become issues. The program below shows a way to minimize space consumption and maximize performance when loading true-color pixrects:

```
/*
 * A program which loads images to the cg8 frame buffer using both pixrect
 * and mmap.  pr_load() is not used, it consumes too much memory resources.
 * We actually open the image file, mmap it into user's addressing space
 * and access it directly.
 */

/*
 * This program is practically:
 *  file_pixrect = pr_load(file);
 *  pr_rop(screen, ...., file_pixrect, ...);
 * only fancier.
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <pixrect/pixrect_hs.h>

Pixrect        *pr_cg8;

main (argc, argv)
    int             argc;
    char           *argv[];

{
    char           *fname;
    FILE           *fp;
    struct stat     st;
    char           *prog;
    char            groups[PIXPG_24BIT_COLOR + 1];
    u_char         *fptr;          /* file pointer */

    prog = *argv++;
    if ((pr_cg8 = pr_open ("/dev/cgeight0")) == NULL ||
    (pr_cg8 = pr_open ("/dev/fb")) == NULL) {
    fprintf (stderr, "%s: fail to open cgeight0 or fb0, prog);
    return 1;
    }

    pr_available_plane_groups (pr_cg8, PIXPG_24BIT_COLOR + 1, groups);
    if (!groups[PIXPG_24BIT_COLOR]) {
    fprintf (stderr, "%s: Not a 24-bit framebuffer0, prog);
    return 1;
    }
    while (--argc > 0 && *argv != NULL) {
    fname = *argv++;
    if (fname &&
        (strlen (fname) <= 0 || (fp = fopen (fname, "r")) == NULL)) {
        pr_close (pr_cg8);
        fprintf (stderr, "%s: fail to open %s0, prog, fname);
        continue;
    }
    if (fstat (fileno (fp), &st) < 0 ||
        (fptr = (u_char *) mmap (0, st.st_size,
              PROT_READ, MAP_SHARED, fileno (fp), 0)) == NULL) {
        fprintf (stderr, "%s: fail to map %s0, prog, fname);
        fclose (fp);
        continue;
    }
    im_load (fptr);
    munmap (fptr, st.st_size);
    fclose (fp);
    }
    pr_close (pr_cg8);
    return;
}

/* loading image from file */
im_load (fptr)
    u_char         *fptr;
{
    struct rasterfile *rh;
    struct pr_pos   off;

    rh = (struct rasterfile *) fptr;
    fptr += sizeof (struct rasterfile);/* lseek */
    off.x = (pr_cg8->pr_size.x - rh->ras_width) / 2;
    off.y = (pr_cg8->pr_size.y - rh->ras_height) / 2;

    if (rh->ras_magic == RAS_MAGIC) {
    switch (rh->ras_depth) {
    case 32:
    case 24:
    case 8:
        load_32 (rh, off, fptr);
        break;
    case 1:
        load_1 (rh, off, fptr);
        break;
    default:
        break;
    }
    }
}

static
load_32 (rh, off, fptr)
```

```
    struct rasterfile *rh;
    struct pr_pos   off;
    u_char          *fptr;

{
    u_char          *r,
                    *g,
                    *b;
    int              maplen;
    register int     linesize;
    register int     fb_increment;
    int              i,
                     j,
                     pixel_size;
    register u_char *im_buffer;
    union fbunit    *fb_image;
    union fbunit    *fb_ptr;

    /*
     * Clear the overlay and enable plane.  Black out the frame buffer and
     * open the hole on the enable plane to let the frame buffer "see thru".
     */
    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY);
    pr_rop (pr_cg8, 0, 0, pr_cg8->pr_size.x, pr_cg8->pr_size.y,
        PIX_SRC | PIX_COLOR (1), 0, 0, 0);
    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY_ENABLE);
    pr_rop (pr_cg8, 0, 0, pr_cg8->pr_size.x, pr_cg8->pr_size.y,
        PIX_SRC | PIX_COLOR (1), 0, 0, 0);
    pr_set_plane_group (pr_cg8, PIXPG_24BIT_COLOR);
    pr_rop (pr_cg8, 0, 0, pr_cg8->pr_size.x, pr_cg8->pr_size.y,
        PIX_SRC, 0, 0, 0);
    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY_ENABLE);
    pr_rop (pr_cg8, off.x, off.y, rh->ras_width, rh->ras_height,
        PIX_SRC, 0, 0, 0);

    /* ready to show the image */
    pr_set_plane_group (pr_cg8, PIXPG_24BIT_COLOR);
    if (rh->ras_maptype == RMT_EQUAL_RGB) {
    maplen = rh->ras_maplength / 3;
    r = (u_char *) fptr;
    fptr += maplen;
    g = (u_char *) fptr;
    fptr += maplen;
    b = (u_char *) fptr;
    fptr += maplen;
    if (rh->ras_depth != 8)
        pr_putlut (pr_cg8, 0, maplen, r, g, b);
    }


    /*
     * Get the pointer to the frame buffer, offset it to the right starting
     * pixel.
```

```
      */
     fb_image = (union fbunit *) mprp_d (pr_cg8)->mpr.md_image;
     pixel_size = rh->ras_depth / 8;    /* amount to increment the file
                          * pointer */

     linesize = mpr_linebytes (rh->ras_width, rh->ras_depth);
     fb_increment = mprp_d (pr_cg8)->mpr.md_linebytes / sizeof (union fbunit);
     if (off.y >= 0)
     fb_image += off.y *
         (mprp_d (pr_cg8)->mpr.md_linebytes / sizeof (union fbunit));
     else
     fptr -= off.y * linesize;
     if (off.x >= 0)
     fb_image += off.x;
     else
     fptr -= pixel_size * off.x;

     for (j = 0; j < rh->ras_height && j < pr_cg8->pr_size.y;
      j++, fptr += linesize, fb_image += fb_increment) {
     im_buffer = fptr;
     fb_ptr = fb_image;
     for (i = 0; i < rh->ras_width && i < pr_cg8->pr_size.x;
          i++, im_buffer += pixel_size, fb_ptr++) {
         switch (rh->ras_depth) {
         case 32:
         fb_ptr->packed = ((union fbunit *) im_buffer)->packed;
         break;
         case 24:{
             union fbunit    store;

             store.channel.B = im_buffer[0];
             store.channel.G = im_buffer[1];
             store.channel.R = im_buffer[2];
             fb_ptr->packed = store.packed;
             break;
         }
         case 8:{
             union fbunit    store;

             store.channel.R = r[*im_buffer];
             store.channel.G = g[*im_buffer];
             store.channel.B = b[*im_buffer];
             fb_ptr->packed = store.packed;

         }
         }
     }
     }
}


#define SMALLER(src, dst, w) \
        (src->pr_size.w < dst->pr_size.w ? src->pr_size.w : dst->pr_size.w)
```

```
static
load_1 (rh, off, fptr)
    struct rasterfile *rh;
    struct pr_pos   off;
    u_char          *fptr;
{
    Pixrect         *mono;
    struct pr_pos   dup;
    struct pr_pos   small;

    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY);
    pr_rop (pr_cg8, 0, 0, pr_cg8->pr_size.x, pr_cg8->pr_size.y,
        PIX_SRC | PIX_COLOR (1), 0, 0, 0);
    fptr += rh->ras_maplength;
    mono = mem_point (rh->ras_width, rh->ras_height, rh->ras_depth, (short *)
              fptr);

    dup.x = dup.y = 0;
    if (off.x < 0)
    dup.x = -off.x, off.x = 0;
    if (off.y < 0)
    dup.y = -off.y, off.y = 0;
    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY_ENABLE);
    pr_rop (pr_cg8, off.x, off.y,
        SMALLER (mono, pr_cg8, x),
        SMALLER (mono, pr_cg8, y),
        PIX_SRC | PIX_COLOR (1), 0, 0, 0);
    pr_set_plane_group (pr_cg8, PIXPG_OVERLAY);
    small.x = SMALLER (mono, pr_cg8, x);
    small.y = SMALLER (mono, pr_cg8, y);
    pr_rop (pr_cg8, off.x, off.y,
        small.x, small.y,
        PIX_SRC, mono, dup.x, dup.y);
    pr_destroy (mono);
}
```

SunView Example Code

```
/*
 * This sample Sunview program shows how to use rops, vectors and pixels.
 */

#include <stdio.h>
#include <suntool/sunview.h>
#include <suntool/canvas.h>

Frame           frame;                          /* Base frame. */
Canvas          canvas;
Pixwin         *pwin;
    char          **argv;
{
    frame = window_create (NULL, FRAME,
                           FRAME_LABEL, argv[0],
                           FRAME_ARGC_PTR_ARGV, &argc, argv,
                           0);
    if (frame == NULL) {
        fprintf (stderr, "%s: Failed to create frame.0, argv[0]);
        exit (1);
    }

    /* Open the canvas. */
    canvas = window_create (frame, CANVAS,
                            CANVAS_RETAINED, FALSE,
                            CANVAS_REPAINT_PROC, repaint_canvas,
                            CANVAS_COLOR24, TRUE,
                            0);
    if (canvas == NULL) {
        fprintf (stderr, "%s: Failed to create canvas. 0, argv[0]);
        exit (1);
    }

    /* Get the pixwin associated with this canvas. */
    pwin = (Pixwin *) window_get (canvas, WIN_PIXWIN);
    if (pwin == NULL) {
        fprintf (stderr, "%s: Failed to get pixwin.0, argv[0]);
        exit (1);
    }

    window_main_loop (frame);
}
```

```
void
repaint_canvas ()
{
    int             i;                  /* Loop index. */
    int             offset = 200;       /* To move ramps toward center of */
                                        /* window. */

    /* Rop a white block (proves that r, g and b work). */
    pw_rop (pwin, 0, 0, 100, 100, (PIX_SRC | PIX_COLOR (0x00ffffff)), 0, 0, 0);

    /* Draw some ramps in different colors. */
    for (i = 0; i < 256; i++) {
        pw_vector (pwin, i + offset, 100, i + offset, 110, PIX_SRC, (i << 0));
        pw_vector (pwin, i + offset, 112, i + offset, 120, PIX_SRC, (i << 8));
        pw_vector (pwin, i + offset, 122, i + offset, 130, PIX_SRC, (i << 16));
    }

    /* Spit out some pixels so we know that they work. */
    pw_put (pwin, 500, 500, 0x00ff0000);        /* red */
    pw_put (pwin, 500, 501, 0x00ff0000);
    pw_put (pwin, 500, 502, 0x00ff0000);
    pw_put (pwin, 501, 500, 0x00ff0000);
    pw_put (pwin, 501, 501, 0x00ff0000);
    pw_put (pwin, 501, 502, 0x00ff0000);
    pw_put (pwin, 504, 500, 0x0000ff00);        /* green */
    pw_put (pwin, 504, 501, 0x0000ff00);
    pw_put (pwin, 504, 502, 0x0000ff00);
    pw_put (pwin, 505, 500, 0x0000ff00);
    pw_put (pwin, 505, 501, 0x0000ff00);
    pw_put (pwin, 505, 502, 0x0000ff00);
    pw_put (pwin, 508, 500, 0x000000ff);        /* blue */
    pw_put (pwin, 508, 501, 0x000000ff);
    pw_put (pwin, 508, 502, 0x000000ff);
    pw_put (pwin, 509, 500, 0x000000ff);
    pw_put (pwin, 509, 501, 0x000000ff);
    pw_put (pwin, 509, 502, 0x000000ff);
}
```

Sun-2 and Sun-3 Assembler

This section describes changes to the *Assembly Language Reference*, Part Number 800-1773, that supports the MC68030 processor included in SunOS release 4.0.3 `as` assembler for the Sun-2 and Sun-3 workstations.

For more information about the MC68030, see the *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, Motorola Inc. document MC68030UM/AD, or later.

New Control Register Support

The MC68030 has six control registers not included in the MC68020 or MC68010. The registers are named `crp`, `srp`, `tc`, `tt0`, `tt1`, and `psr`.

Replace the table on page 43 of the *Assembly Language Reference* with this table containing the new register names:

Name	*Register*
`sp`	the stack pointer, which is equivalent to `a7`
`sr`	the status register
`cc`	the condition codes of the status register
`usp`	the user mode stack pointer
`pc`	the program counter
`sfc`	the source function code register
`dfc`	the destination function code register
`fpcr`	the floating-point control register
`fpsr`	the floating-point status register
`fpiar`	the floating-point instruction address register
`crp`	64-bit CPU root pointer
`srp`	64-bit supervisor root pointer
`tc`	32-bit translation control register
`tt0`	32-bit transparent translation register
`tt1`	32-bit transparent translation register
`psr`	MMU status register

New Processor Instruction Support

The new MC68030 instructions supported by the SunOS Release 4.0.3 `as` are: `pflush`, `pload`, `pmove`, `ptest`, and their variants. Note that all the new instructions are privileged.

The following table lists the instructions specific to the MC68030 as supported by `as`. Append it to the end of Table B-1 starting on page 60 in the *Assembly Language Reference*.

NOTE `as` *has no explicit* `-030` *flag: the new instructions are always recognized and assembled. Obviously, this would create problems with code run on non-MC68030 machines.*

Table 3 *List of MC68030 Instruction Codes*

Mnemonic	Operation Name	Syntax	Processor
pflusha	flush all ATC entries	pflusha	MC68030
pflush	flush ATC entry	pflush *#data*,*#data*	MC68030
		pflush d*n*,*#data*	MC68030
		pflush d*n*,*#data*,*ea*	MC68030
		pflush sfc,*#data*	MC68030
		pflush sfc,*#data*,*ea*	MC68030
		pflush dfc,*#data*,a*x*@(8,d7:1)	MC68030
		pflush dfc,*#data*,a*x*@	MC68030
ploadr	load ATC entry	ploadr *#data*,2:1	MC68030
		ploadr d*n*,a*x*@(8)	MC68030
		ploadr sfc,a*x*@(8,d7:1)	MC68030
		ploadr dfc,a*x*@	MC68030
ploadw		ploadw *#data*,2:1	MC68030
		ploadw d*n*,a*x*@(8)	MC68030
		ploadw sfc,a*x*@(8,d7:1)	MC68030
		ploadw dfc,a*x*@	MC68030
pmove	move to/from MMU registers	pmove crp,a*x*@	MC68030
		pmove srp,a*x*@	MC68030
		pmove tc,a*x*@	MC68030
		pmove psr,a*x*@	MC68030
		pmove tt0,a*x*@	MC68030
		pmove tt1,a*x*@	MC68030
		pmove a*x*@,crp	MC68030
		pmove a*x*@,srp	MC68030
		pmove a*x*@,tc	MC68030
		pmove a*x*@,psr	MC68030
		pmove a*x*@,tt0	MC68030
		pmove a*x*@,tt1	MC68030
ptestr	test logical address	ptestr *#data*,2:1,*#data*	MC68030
		ptestr d*n*,a*x*@(8),*#data*	MC68030
		ptestr sfc,a*x*@(8,d7:1),*#data*,a3	MC68030
		ptestr dfc,a*x*@,*#data*,a4	MC68030
ptestw		ptestw *#data*,2:1,*#data*	MC68030
		ptestw d*n*,a*x*@(8),*#data*	MC68030
		ptestw sfc,a*x*@(8,d7:1),*#data*,a3	MC68030
		ptestw dfc,a*x*@,*#data*,a4	MC68030
pmoveflush	move to/from MMU register, flush ATC	pmoveflush crp,a*x*@	MC68030
		pmoveflush srp,a*x*@	MC68030
		pmoveflush tc,a*x*@	MC68030
		pmoveflush psr,a*x*@	MC68030
		pmoveflush tt0,a*x*@	MC68030
		pmoveflush tt1,a*x*@	MC68030

Table 3 *List of MC68030 Instruction Codes— Continued*

Mnemonic	*Operation Name*	*Syntax*	*Processor*
		`pmoveflush ax@,crp`	MC68030
		`pmoveflush ax@,srp`	MC68030
		`pmoveflush ax@,tc`	MC68030
		`pmoveflush ax@,psr`	MC68030
		`pmoveflush ax@,tt0`	MC68030
		`pmoveflush ax@,tt1`	MC68030

Device Drivers

This section corrects portions of *Writing Device Drivers* , Part Number 800-1780-10, for SunOS Release 4.0, 4.0.1, and 4.0.3.

`rmalloc()`

Pages 35 (bottom) and 377 refer to `rmalloc()` and `iopbmap`. The statement "this will get a small block of memory back from the beginning of DVMA space" does not describe how this space is accessed.

The `iopbmap` is a byte-aligned table. The address it returns is not aligned on a long word boundary. If a non-aligned address is accessed, a panic may result. Callers of `rmalloc()` should ask for a few bytes of memory more than they need, and round up the address to a full word boundary if necessary. This applies to both Sun-3's and Sun-4's, but it is more critical to Sun-4's, since they can only address using full word alignment.

Sun386i Interrupt

On the bottom of page 352 and the top of page 353, it states that `xxintr()` receives by default the unit number of the device that interrupted it, or something else by changing the value in `md_intr->v_vptr`.

This does not apply to the Sun386i. The Sun386i receives two arguments. The first is the current priority level (`cpl`) and the second is the interrupt request (`irq`). The irq is hardwired so it cannot be changed. The interrupt routine can never receive the unit number. The unit number can be obtained by saving the interrupt request channel (board level + 8) at attach time and then figure out which device received the interrupt at interrupt time.

`tty_std`(4M)

On page 164, in Figure 9-2, the names in the left column: `tty_compat.4m`, `tty_std.4m`, and `nit.4m` are System V names. These names are now `ttcompat.4m`, `ldterm.4m`, and `nit.4p` respectively. These same changes should also be made on pages 311 and 312.

`cdevsw`

On the bottom of page 45, the `cdevsw` structure is listed with each element. The two elements `d_stop` and `d_ttys` no longer exist in SunOS and should not be in the table.

`kmem_alloc`

On page 369, the `kmem_alloc()` description says the routine calls `panic()` if its request cannot be satisfied. This was true in SunOS 3.X, but not for SunOS 4.0 and above. In SunOS 4.0 and above, `NULL` pointer is returned if the space cannot be allocated.

`skioctl`

On the bottom of page 142 the elements of the `cdevsw` are described. There are two routines shown that are no longer used. References to `xxstop()` and "a `tty` structure pointer" should not be there. See `cdevsw` explanation above. The description of the `cdevsw` entries in the box on page 142 should also be changed to reflect the new order. The three lines that say

```
skopen, skclose, skread, skwrite,
nodev, nodev, nodev, 0,
seltrue, skmmap
```

should occur in the following order. References to the `ioctl` routine should include the code from page 135 in the previous chapter, and references to the `select` routine should include the code from page 132. The new list should be:

```
skopen, skclose, skread, skwrite,
skioctl, nodev, skselect, skmmap
```

`selwait`

On pages 133 and 134, there are references to `selwait()` making it look like it is a function. `Selwait` is an integer, so there should be no parenthesis. This reference is at the top and bottom of page 133 and the top of page 134. In addition, the index references on page 449 and 451 should not have parenthesis.

Ramdisk

Chapter 8 (pages 151 - 156) describes how to implement a ramdisk pseudo-device driver. This example is based on SunOS 3.5 and does not work on SunOS 4.0. If this program is run under SunOS 4.0 it may cause your system to crash.

Ramdisk node

In the box on page 154 is a list of functions for the `cdevsw`. As described in the `cdevsw` section above, this list should have two less entries. The list should look like this:

```
ramopen, nulldev, ramread, ramwrite,
nodev, nulldev, seltrue, nodev
```

On page 154 the `mknod` example references `ram0c` as the device name. The correct command should be:

```
/etc/mknod ram0 b 8 0
/etc/mknod rram0 c 30 0
```

On the top of page 155 the reference to device name `/dev/ram0c` should instead be `/dev/ram0`.

`strlog()`

On page 337 is a description of the routine `strlog()`. Although this appears in the manual, it is not a Sun-supported routine and is not recommended to use.

`tty` structures

In Chapter 3 on page 45, at the end of the second paragraph, the statement "For terminals, the `cdevsw` structure also contains a pointer to an array of `tty` structures associated with the driver." should be deleted.

`tty` structure

In Chapter 3 on the top of page 46, the first paragraph "Only teletype-like devices (such as the the console driver, the `mti` driver, and the `zs` driver) use the `tty` structure. All other devices set it to zero." should be deleted.

`cdevsw` structure

In Chapter 3 on page 46, the first box shows the `cdevsw` structure. Some of the names have been changed. The original routines are:

```
cgoneopen,  cgoneclose, nodev,  nodev,  /*14*/
cgoneioctl, nodev,  nodev,  0,
seltrue,    cgonemmap,
```

which should be changed to:

```
cgoneopen,  cgoneclose, nodev,      nodev,      /*14*/
cgoneioctl, nodev,      seltrue,    cgonemmap,
0,      spec_segmap,
```

`md_driver`

In Chapter 3 on page 52, the last word on the page ''definition'' should be ''declaration.''

kernel

In Chapter 4 on page 63, the statement ''kernel is *monolithic monitor* type of operating system'' should have the word ''a'' inserted resulting in ''kernel is a *monolithic monitor* type of operating system''

header file

In Chapter 6 on page 114, the box has a series of include statements. They should be changed from

```
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/file.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/uio.h"
#include "../machine/psl.h"
#include "../sundev/mbvar.h"
```

to

```
#include <sys/param.h>
#include <sys/buf.h>
#include <sys/file.h>
#include <sys/dir.h>
#include <sys/user.h>
#include <sys/uio.h>
#include <machine/psl.h>
#include <sundev/mbvar.h>
```

`uio.h`

In Chapter 6 on page 120, the last paragraph makes reference to an include file as `/usr/include/sys/uio.h` which should be changed to `<sys/uio.h>`.

`conf.c`

In Chapter 7 on page 139, the first square item states that `conf.c` is ''a C-language source-code file which contains the definitions of the switches''. Definitions should really be changed to ''a C-language source-code file which contains the default initializations of the switches''

`ram.c`

In Chapter 8 on page 152, the box contains include statements for the driver. These statements should be changed from:

```
#include "../h/param.h"      /* Includes "../h/types.h" */
#include "../h/errno.h"
#include "../h/uio.h"
#include "../h/buf.h"
```

to:

```
#include <sys/param.h>       /* Includes <sys/types.h> */
#include <sys/errno.h>
#include <sys/uio.h>
#include <sys/buf.h>
```

alternatives

In Chapter 9 on page 191, the end of the second paragraph states" ''interfaces to the two the non-802.2 drivers and the IP multiplexor.''. There is an extra ''the'' that should be deleted, making ''the two non-802.2 drivers and the IP multiplexor.''.

`putctl1`

In Appendix A on page 335, the putctl description refers to the box before it. The statement ''section, below). On successful completion'' should be ''section, above). On successful completion''.

`testb`

In Appendix A on page 337, the box for `testb()` has a C statement

```
int size, pri;
```

that should be

```
register size;
uint pri;
```

boottime

In Appendix A on page 341, under `kernel.h` is the statement

```
struct timeval boottime /* time since system came up */hz
```

that should be

```
struct timeval boottime /* time since system came up */
```

QUEUE

In Appendix A on page 342, the square item at the bottom of the page has a period after the word QUEUE which should be a comma.

Notes