Starbase Technical Addendum for the July, 1997 Workstation ACE for 10.20 HP-UX

HP 9000 Series 700 and 800 Computers

HEWLETT PACKARD

HP Part No. B2355-90143
Printed in USA July 1997

Edition 1

FINAL TRIM SIZE : 7.5 in x 9.0 in

Notices

Printing History

New editions of this manual will incorporate all material updated since the previous edition. This manual's printing date and part number change when a new edition is printed.

July 1997 . . . Edition 1. This manual is valid for the HP-UX 10.20 release on all HP 9000 Series 700 and 800 Computers, plus the July, 1997 Workstation ACE for 10.20 HP-UX on all HP 9000 Series 700 Computers. This edition of the manual includes device information for the HP VISUALIZE-EG, HP VISUALIZE-8, HP VISUALIZE-24, HP VISUALIZE-48, HP VISUALIZE-48XP, the HP VISUALIZE-FX family of devices, and HP VMX. Note that Starbase reference pages for new and updated features are also part of this addendum.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Contents

Figures

Tables

1 Introduction

The *Starbase Technical Addendum for HP-UX 10.X* are addenda to the 10.0 versions of the *HP-UX Starbase Device Drivers Manual* and the *Starbase Reference*. This addendum contains the following chapters and appendix:

Chapter 1	"Introduction" is a brief introduction to this addendum. It provides information on chapter contents and manual conventions.
Chapter 2	"The HCRX Family of Devices" — This chapter describes the HCRX family of graphics devices, and the addition of the HP VISUALIZE-8, HP VISUALIZE-24, HP VISUALIZE-EG, and Dual HP VISUALIZE-EG cards.
Chapter 3	"The HP VISUALIZE-48 and HP VISUALIZE-48XP Devices" — This chapter describes the functionality of these graphics devices.
Chapter 4	"The HP VISUALIZE-FX Family of Devices" — This chapter describes the functionality of the HP VISUALIZE-FX^2, HP VISUALIZE-FX^4, and HP VISUALIZE-FX^6 devices.
Chapter 5	"HP Virtual Memory and X (VMX)" — This chapter describes how the `hpvmx` device driver works.
Chapter 6	"Gescapes" — This chapter describes new gescapes for the HP-UX 10.X releases of Starbase.
Appendix A	"Starbase Reference Pages" — This appendix contains new and updated reference pages for the HP-UX 10.X releases of Starbase.

Conventions

This section covers documentation conventions and the conventions used to set environment variables.

Document Conventions

⟨*italics*⟩	Italic type enclosed in angle brackets indicates parameters that you will supply. This same convention also applies to the actual path names of directories. Note that the path names given in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.
⋮	The vertical ellipsis indicates that irrelevant parts of a program example or illustration have been omitted.
`computer font`	This constant-width, computer-type font indicates verbatim entries that you make on the command line or as program text including routine names, parameters, and file or directory names.

Environment Variable Conventions

In this document, you will be given examples that set various environment variables using the Korn shell.

An environment variable can be set using any one of three shell environments: Korn shell, POSIX shell, and C shell. The Korn and POSIX shells set environment variables in the same manner; however, the C shell uses a different command.

Setting and Unsetting Environment Variables

To set an environment variable (*env_var*) to a particular value (*value*), or to unset an environment variable (*env_var*), use the information provided in Table 1-1.

Table 1-1.
Shell Commands for Setting and Unsetting Environment Variables

Korn and POSIX Shells	C Shell
To set: `export` ⟨*env_var*⟩`=`⟨*value*⟩	To set: `setenv` ⟨*env_var*⟩ ⟨*value*⟩
To unset: `unset` ⟨*env_var*⟩	To unset: `unsetenv` ⟨*env_var*⟩

Setting PATH and MANPATH

In this document, you will be given examples that refer to various executables and on-line reference pages. To conveniently access these executables and online reference pages, your `PATH` and `MANPATH` environment variables must include the appropriate Starbase and graphics directories. The following commands will add the necessary directories. (The actual path name of the directory given in angle brackets depends on the file system structure; see the *Graphics Administration Guide* for details.)

```
PATH=$PATH:⟨starbase⟩/bin:⟨common⟩/bin
MANPATH=$MANPATH:⟨starbase⟩/share/man
export PATH MANPATH
```

HP CDE and HP VUE

Hewlett-Packard is in the process of transitioning to a standard user environment. Two user environments are shipped with HP-UX 10.20: HP VUE and HP CDE (Common Desktop Environment). Starting with HP-UX 10.20, HP CDE will be the default user environment. HP VUE will still be available with HP-UX 10.20, but will not be available in future HP-UX releases. See the *Common Desktop Environment User's Guide* for more information on HP CDE.

From a 3D graphics point of view, the change in user environments should have no effect.

Gamma Correction

A new gamma correction tool is available with July, 1997 Workstation ACE for 10.20 HP-UX. See the *Graphics Administration Guide* for more information.

Texture Mapping

HP-UX releases 10.10 and later support Starbase texture mapping. Here is a list of Starbase texture mapping routines:

- `tm_activate_texture`(3G)
- `tm_activate_bf_texture`(3G)
- `tm_close_texture`(3G)
- `tm_edit_texture`(3G)
- `tm_filter_texels`(3G)
- `tm_load_texels`(3G)
- `tm_open_texture`(3G)
- `tm_resource_hints`(3G)
- `tm_unload_texels`(3G)
- `tm_view_orientation`(3G)
- `tmc_partial_polygon_with_data3d`(3G)
- `tmc_polygon_with_data3d`(3G)
- `tmc_polyhedron_with_data`(3G)

- `tmc_quadrilateral_mesh_with_data`(3G)
- `tmc_triangular_strip_with_data`(3G)

See the reference pages in Appendix A for more information on these routines.

2

The HCRX Family of Devices

This chapter describes the HCRX family of graphics devices, and the addition of the HP VISUALIZE-8, HP VISUALIZE-24, HP VISUALIZE-EG, and Dual HP VISUALIZE-EG cards.

The `hphcrx` driver supports the HCRX family of graphics devices. These devices are all similar, and have different levels of hardware support for the following operations:

- Generating vectors
- Write-enabling planes
- Writing pixels to the frame buffer with a given replacement rule
- Moving a block of pixels from one place in the frame buffer to another
- Using bank-select and double-buffering per window
- Flat shaded rectangles
- Dithering and HP Color Recovery
- Overlay plane transparency (X Windows)
- Geometry transformations (with PowerShade)
- Lighting calculations (with PowerShade)
- Interpolated shading (with PowerShade).

The `hphcrx` device driver supports the graphics devices described in the subsequent section.

hphcrx Devices

Note Raw-mode graphics support is *not* available on any HCRX graphics device. You must display your Starbase applications in an X11 window or windows. For information on using Starbase with X11, read the chapter "Using Starbase with the X Window System" in the *Starbase Graphics Techniques* manual.

The `hphcrx` device driver supports the configurations shown in Table 2-1.

Table 2-1. hphcrx Device Driver Support

Graphics Device	Number of Image Planes	Number of Overlay Planes[1]	Hardware Accelerator	Resolution
HP VISUALIZE-EG	8	0	No	1280×1024
HP VISUALIZE-EG with add-on memory	8 or 8/8	8	No	1280×1024
HCRX-8	8 or 8/8	8	No	1280×1024
HCRX-8Z	8 or 8/8	8	Yes	1280×1024
HP VISUALIZE-8	8 or 8/8	8	Yes[2]	1280×1024
HCRX-24	24, 12/12, or 8/8	8	No	1280×1024
HCRX-24Z	24, 12/12, or 8/8	8	Yes	1280×1024
HP VISUALIZE-24	24, 12/12, or 8/8	8	Yes[2]	1280×1024

1 All Starbase graphics rendering to the overlay planes is done by the VMX or SOX11 device drivers.

2 This graphics device includes a geometry accelerator.

The HCRX-8, HCRX-8Z, HP VISUALIZE-EG with add-on memory, and HP VISUALIZE-8 support 8 planes single-buffered or 8/8 planes double-buffered. The HCRX-24, HCRX-24Z, and HP VISUALIZE-24 support 8 planes single-buffered, 8/8 planes double-buffered, 12 planes single-buffered, 12/12 planes double-buffered, or 24 planes single-buffered. The HP VISUALIZE-EG with no add-on memory supports 8 planes single-buffered. You must install the add-on HP VISUALIZE-EG memory to add support for 8/8 double-buffering and 8 overlay planes.

In order to reduce flickering, these graphics devices refresh the attached CRT displays at a minimum rate of 72 Hz.

All configurations support 1280×1024 pixel color displays, two hardware color maps in the image planes, and two hardware color maps in the overlay planes when overlay planes are available.

PowerShade

The 3D surfaces software, PowerShade, works with all of the `hphcrx` devices. Note that PowerShade only works with the `hphcrx` driver in the image planes. PowerShade support in the overlay planes is provided by the VMX driver. For information on this driver, see the "Virtual Memory and X" chapter in this Addendum.

In order to use VMX with PowerShade on any graphics system, you must install the PowerShade software. See the *Graphics Administration Guide* for more details.

For More Information

Information in this section is device-specific. For more detailed information on graphics programming and X windows, please refer to the noted documents:

- See the *Starbase Graphics Techniques* manual for general Starbase programming information.
- Refer to the *Graphics Administration Guide* to read about linking shared or archive libraries, path naming conventions, X windows, completing installation, and setting up graphics devices.

Device Descriptions

HP VISUALIZE-EG Card

The HP VISUALIZE-EG card without the add-on HP VISUALIZE-EG memory has two hardware color maps in the image planes. With the add-on HP VISUALIZE-EG memory, HP VISUALIZE-EG has two hardware color maps in the image planes and two hardware color maps in the overlay planes. It supports HP Color Recovery, as explained in the section "HP Color Recovery Technology" in this chapter.

This graphics card provides a display that is 1280×1024 pixels. There is no offscreen memory in the frame buffer. Note that the 1600×1200 resolution mode is not supported.

The HP VISUALIZE-EG supports 8 planes single-buffered without the add-on HP VISUALIZE-EG memory. Or, if you have the add-on HP VISUALIZE-EG memory, 8/8 double-buffered is supported. After you add the add-on HP VISUALIZE-EG memory, you must configure the memory the first time you `reboot` your system.

The PowerShade software adds support for lighting, shading, and Z-buffering in all windows.

The only X server mode supported is combined mode. For information on supported X server modes, read the section "Supported X Server Modes" in the *Graphics Administration Guide*.

HP VISUALIZE-EG supports all Starbase color map modes (CMAP_MONOTONIC, CMAP_FULL, and CMAP_NORMAL).

Dual HP VISUALIZE-EG Card and Multi-Display Support

The Dual HP VISUALIZE-EG card has two HP VISUALIZE-EG devices on the same card and each device provides a separate but identical set of HP VISUALIZE-EG features.

The Dual HP VISUALIZE-EG card has two video output connections that provide multi-display support. Multi-display support allows you to configure two or more heterogeneous or homogeneous displays to the same workstation. You can have a maximum of four displays connected to your workstation.

For information on multi-display support and how to configure multiple displays, read the *Graphics Administration Guide*.

Add-on HP VISUALIZE-EG Memory

The add-on HP VISUALIZE-EG memory is a card that provides an extra two megabytes of memory to the HP VISUALIZE-EG to which it is connected. This added memory provides 8/8 double-buffering in the image planes and 8 overlay planes when used with Starbase.

This graphics card provides a display that is 1280×1024 pixels. There is no offscreen memory in the frame buffer. Note that the 1600×1200 resolution mode is not supported.

For information on connecting the add-on HP VISUALIZE-EG memory card(s) to your HP VISUALIZE-EG card or Dual HP VISUALIZE-EG card, read the *User's Guide* that comes with your graphics device.

HCRX-8

The HCRX-8 graphics device has two hardware color maps in the image planes and two hardware color maps in the overlay planes. It supports HP Color Recovery, as explained in the section "HP Color Recovery Technology" in this chapter.

This display is 1280x1024 pixels with two banks of eight image planes and eight overlay planes. There is no offscreen memory in the frame buffer.

The HCRX-8 device supports 8 planes single-buffered or 8/8 double-buffered.

The PowerShade software adds support for lighting, shading, and Z-buffering in all windows.

The only X server mode supported is combined mode. For information on supported X server modes, read the section "Supported X Server Modes" in the *Graphics Administration Guide*.

The HCRX-8 device supports all Starbase color map modes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, and `CMAP_FULL`).

There are two hardware color maps available for use with the image planes. At most, one of the two color maps will be dedicated for use by DirectColor or TrueColor visuals. The other one will be shared by all indexed color graphics windows.

In addition to the two hardware color maps in the image planes, there are two hardware color maps in the overlay planes. One of the hardware color maps has the default X11 color map permanently installed in it. This is done to avoid *technicoloring* your X11 and HP CDE backgrounds and applications.

HCRX-8Z

This device is a superset of the HCRX-8 graphics device.

The HCRX-8Z graphics device includes an accelerator that attaches to the HCRX-8 device to accelerate rendering into the frame buffer. The HCRX-8Z accelerator has a dedicated 24-bit Z-buffer. The **hphcrx** driver automatically uses the HCRX-8Z accelerator if it is present. The primary use of the HCRX-8Z accelerator is for 3D solids modeling, including drawing Starbase polygons, rectangles, triangle strips, quadrilateral meshes, and spline surfaces.

The HCRX-8Z device supports all Starbase color map modes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, and `CMAP_FULL`). Note that the HCRX-8Z does not dither when shading in `CMAP_MONOTONIC` mode.

HP VISUALIZE-8

This device is a superset of the HCRX-8 graphics device.

The HP VISUALIZE-8 graphics device includes a geometry accelerator that attaches to the HCRX-8 device to provide high performance 3D solids modeling and high performance 3D wireframe. Note that this accelerator replaces the HCRX Z-buffer/Accelerator. The HP VISUALIZE-8 geometry accelerator has a dedicated 24-bit Z-buffer. The `hphcrx` driver automatically uses the HP VISUALIZE-8 geometry accelerator if it is present. The primary use of the HP VISUALIZE-8 accelerator is for 3D solids modeling, including drawing Starbase polygons, rectangles, triangle strips, quadrilateral meshes, and spline surfaces.

This device supports all Starbase color map modes (`CMAP_FULL`, `CMAP_NORMAL`, and `CMAP_MONOTONIC`). Note that the HP VISUALIZE-8 does not dither when shading in `CMAP_MONOTONIC` mode.

For more information on the geometry accelerator, read the section "Geometry Accelerator" in this chapter.

HCRX-24

The HCRX-24 graphics device has two hardware color maps in the image planes and two hardware color maps in the overlay planes. It supports HP Color Recovery in all 8 plane visuals, as explained in the section "HP Color Recovery Technology" in this chapter.

The HCRX-24 graphics device has 24 image planes and 8 overlay planes. The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer.

You can render to the image planes in three ways:

- 8-bit indexed color (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`)
- 12-bit direct color (`CMAP_FULL`)
- 24-bit direct color (`CMAP_FULL`)

The three rendering modes are selected on a per-window basis. The mode selected is a function of the depth of the window created and the double-buffer mode.

The PowerShade software adds support for lighting, shading, and Z-buffering in all windows.

In 8-bit indexed mode, each pixel is used as an index into a 256-entry color map. Each entry in the color map provides eight bits per color for each of the red, green and blue components, providing a color palette of over 16 million colors. Double-buffering is achieved by switching between two banks of 8-bit indexes. You can perform 3:3:2 direct color emulation in this mode.

In 12-bit direct color mode, each pixel is represented by four bits per color channel to allow for double-buffering. One buffer resides in the upper 4 bits and the other buffer resides in the lower 4 bits of each color channel. Dithering improves the color resolution.

In 24-bit direct color mode, a pixel is represented by eight bits each for red, green, and blue. Double-buffering is not supported in this mode.

There are two hardware color maps available for use with the image planes. At most, one of the two color maps will be dedicated for use by direct color graphics windows. The other one will be shared by all indexed color graphics windows.

In addition to the two hardware color maps in image planes, there are two hardware color maps in overlay planes. One of the hardware color maps has the default X11 color map permanently installed in it. This is done to avoid *technicoloring* your X11 and HP CDE backgrounds and applications.

The X server works only in combined mode. For information on supported X server modes, read the section "Supported X Server Modes" in the *Graphics Administration Guide*.

HCRX-24Z

This device is a superset of the HCRX-24 graphics device.

The HCRX-24Z includes an accelerator that attaches to the HCRX-24 device to accelerate rendering into the frame buffer. The HCRX-24Z accelerator has a dedicated 24-bit Z-buffer. The `hphcrx` driver automatically uses the HCRX-24Z accelerator if it is present. The primary use of the HCRX-24Z accelerator is for 3D solids modeling, including drawing Starbase polygons, rectangles, triangle strips, quadrilateral meshes, and spline surfaces.

The HCRX-24Z accelerator can render to the image planes on the HCRX-24 in depth 8, depth 12, and depth 24 visuals.

Note that the HCRX-24Z does not dither when shading in `CMAP_MONOTONIC` mode.

HP VISUALIZE-24

This device is a superset of the HCRX-24 graphics device.

The HP VISUALIZE-24 graphics device includes a geometry accelerator that attaches to the HCRX-24 device to provide high performance 3D solids modeling and high performance 3D wireframe. Note that this accelerator replaces the HCRX Z-buffer/Accelerator. The HP VISUALIZE-24 geometry accelerator has a dedicated 24-bit Z-buffer. The `hphcrx` driver automatically uses the HP VISUALIZE-24 geometry accelerator if it is present. The primary use of the HP VISUALIZE-24 accelerator is for 3D solids modeling, including drawing Starbase polygons, polyhedrons, rectangles, triangle strips, quadrilateral meshes, and spline surfaces.

This device supports all Starbase color map modes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, and `CMAP_FULL`). Note that the HP VISUALIZE-24 does not dither when shading in `CMAP_MONOTONIC` mode.

For more information on the geometry accelerator, read the section "Geometry Accelerator" in this chapter.

Overlay Plane Rendering

Either the `hpvmx` or the `sox11` device driver is used for Starbase rendering to the overlay planes. For more information on Starbase rendering to the overlay planes, read the chapters "HP Virtual Memory and X" in this Addendum and "The Starbase-on-X11 Device Driver" in the *Starbase Device Drivers Manual*.

8/8 VM (Virtual Memory) double-buffering is also supported in the overlay planes using the `hpvmx` driver. If an overlay plane window is `gopen`ed with a driver name of `NULL`, the `hpvmx` driver will be used. See the table, "Driver Selection at gopen" in the chapter "HP Virtual Memory and X" in this Addendum for details.

Geometry Accelerator (only HP VISUALIZE-8 and HP VISUALIZE-24)

The HP VISUALIZE-8 and HP VISUALIZE-24 graphics devices include an accelerator for geometry transformation, lighting, and shading of primitives.

The following lists provide information to help you maximize your application performance. The first list describes operations that yield the best performance when using the new accelerators.

- Isotropic modeling transformations
- Lighting, with up to 8 lights of any type
- View clipping
- Perspective and orthographic (parallel) transformations
- Depth cueing
- 3- and 4-sided filled primitives, with or without RGB, alpha, and normal per vertex
- Triangle strips, with or without RGB, alpha, and normal per vertex
- 2D and 3D polylines

The following operations are less efficient because Starbase must perform extra calculations before using the geometry accelerator:

- Non-convex polygons with more than 4 vertices
- Polyhedrons with move/draw flags
- Edged primitives (but not edging with move/draw flags)
- Facet normal lighting
- Facet color

For the following features, PowerShade bypasses the geometry accelerator and partially calculates the results using software:

- Self-intersecting polygons
- Model clipping/capping
- Deformation
- Wide lines
- Backface distinguishing (but not back-face culling)
- Starbase `INT_OUTLINE` interior style
- Circles, ellipses, arcs
- `CMAP_NORMAL` or `CMAP_MONOTONIC` modes
- Picking
- `move3d()/draw3d()`
- Polymarkers
- Rectangles
- Text

Note that since the geometry accelerator only directly handles polygons with 3 or 4 vertices, more complicated polygons are decomposed into triangles. Convex polygons will be decomposed with the most efficiency. Non-convex polygons or fill area sets with only one set will be decomposed with less efficiency. Polyhedrons with move/draw flags will be decomposed, but with a great penalty in execution time. Self-intersecting polygons cannot be decomposed for the geometry accelerator; instead, they are lighted, shaded, and transformed by PowerShade, with only the final rendering steps performed by the HP VISUALIZE-8 and HP VISUALIZE-24 hardware. Since polygons are decomposed into triangles before transformations occur, visual results may differ slightly from previous devices. Non-planar polygons or polygons with greatly differing colors or normals at the vertices will differ more than planar polygons or polygons with more homogeneous vertex data.

Also, note that compound primitives (triangle strips, quadrilateral meshes, polyhedrons, etc.) will perform better than the equivalent discrete polygon calls, since the there is less procedure call overhead.

For more information about specific primitives and their relative speed, read the *Graphics Administration Guide*.

PowerShade, 3D Surfaces Software

PowerShade is an optional software package that supports lighting and shading in graphics applications. It has capabilities for both surface rendering and volumetric rendering, the latter on the HCRX-24, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 only.

In order to use PowerShade on these devices, you must first *install* the PowerShade software. See the *Graphics Administration Guide* for more information.

HP Color Recovery Technology

Starbase automatically uses HP Color Recovery for shaded fill areas (for example, polygons and spline surfaces) in depth 8 image-plane visuals. On HCRX devices, color recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. HP Color Recovery is available on all HCRX graphics devices with depth 8 windows.

There are two components of HP Color Recovery: a different dither cell size (16×2) is used when rendering shaded polygons, and a digital filter is used when displaying the contents of the frame buffer on the screen.

HP Color Recovery is available on the HCRX family of graphics devices whenever you are in `CMAP_FULL` mode and you have used the `INIT` flag in the `gopen`, `shade_mode`, or the `double_buffer` function to initialize color maps. Keep in mind that the default color map mode is `CMAP_NORMAL` for PseudoColor visuals. Therefore, the HP Color Recovery color map will not be downloaded until you call `shade_mode` to set the mode to `CMAP_FULL` and use `INIT`.

HP Color Recovery is available when using either PseudoColor or TrueColor visuals.

HP Color Recovery is enabled in conjunction with a particular X color map that is associated with your window. If that X color map is not currently installed in hardware by your window manager, you will not see the effect of the HP Color Recovery filter.

The HP Color Recovery color map is a READ-ONLY color map. Any attempts to change it will be ignored and no error will be reported.

Note that vectors are almost always dithered, even in an HP Color Recovery window. The only exceptions to this are that Z-buffered and depth-cued vectors are not dithered on the HCRX-8, HCRX-8Z, HCRX-24, and HCRX-24Z.

Under some conditions HP Color Recovery can produce undesirable artifacts in the image. This can also happen with 4×4 dithering, but the artifacts are different. However, images rendered with HP Color Recovery are seldom worse than what dithering produces, and in most cases, HP Color Recovery produces significantly better pictures than dithering. Note that 4×4 dithering, like HP Color Recovery, is available in the `CMAP_FULL` color map mode, but *not* in the `CMAP_NORMAL` color map mode.

HP Color Recovery is available by default. If you wish to disable HP Color Recovery, you can do it in one of three ways:

- Add the screen option `DisableColorRecovery` to your `X*screens` file. Setting this screen option prior to starting up the X server disables HP Color Recovery for *all* applications and any attempts to enable HP Color Recovery will be ignored. Remember, if you set this screen option prior to starting up the X server, you cannot re-enable HP Color Recovery from within an application. To set this screen option before starting the X server, add the following lines to your ⟨*x11-admin*⟩[1]`/X0screens` file:

  ```
  ScreenOptions
     HP_DISABLE_COLOR_RECOVERY
  ```

  or the preferred entry is:

  ```
  ScreenOptions
     DisableColorRecovery
  ```

  and restart HP CDE or X11.

- Export the environment variable `HP_DISABLE_COLOR_RECOVERY` before running your application. Setting this environment variable to any value disables HP Color Recovery for subsequently executed applications. To set this environment variable in your current X11 window, execute this command on the command line before running your application (assuming you are using the Korn shell):

  ```
  export HP_DISABLE_COLOR_RECOVERY=TRUE
  ```

[1] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

- Disable HP Color Recovery programmatically by using the Starbase `gescape` `COLOR_RECOVERY_CONTROL`. For details on this `gescape`, read the subsequent section "Gescapes."

In `CMAP_FULL` shade mode, disabling HP Color Recovery will result in normal dithering of shaded fill areas. HP Color Recovery is not available in any other shade mode.

Gescapes

The `COLOR_RECOVERY_CONTROL` `gescape` can be used to disable HP Color Recovery. Passing it a 0 value in `arg1` will disable HP Color Recovery, a 1 value will enable it (HP Color Recovery is enabled by default). The `arg2` parameter is ignored. The effect of this `gescape` will not take place until the next time you call `shade_mode` or `double_buffer` with the `INIT` flag. For example:

```
gescape_arg  arg1;

/* Disable HP Color Recovery */
arg1.i[0] = 0;
gescape(fildes, COLOR_RECOVERY_CONTROL, &arg1, NULL);
shade_mode(fildes, CMAP_FULL|INIT,0);
```

The Frame Buffer

Physical Address Space

The physical frame buffer is addressed as 2048×1024 bytes. The last 768 bytes of each line of the address space (those to the right of the screen) are not displayed and no memory exists in those areas.

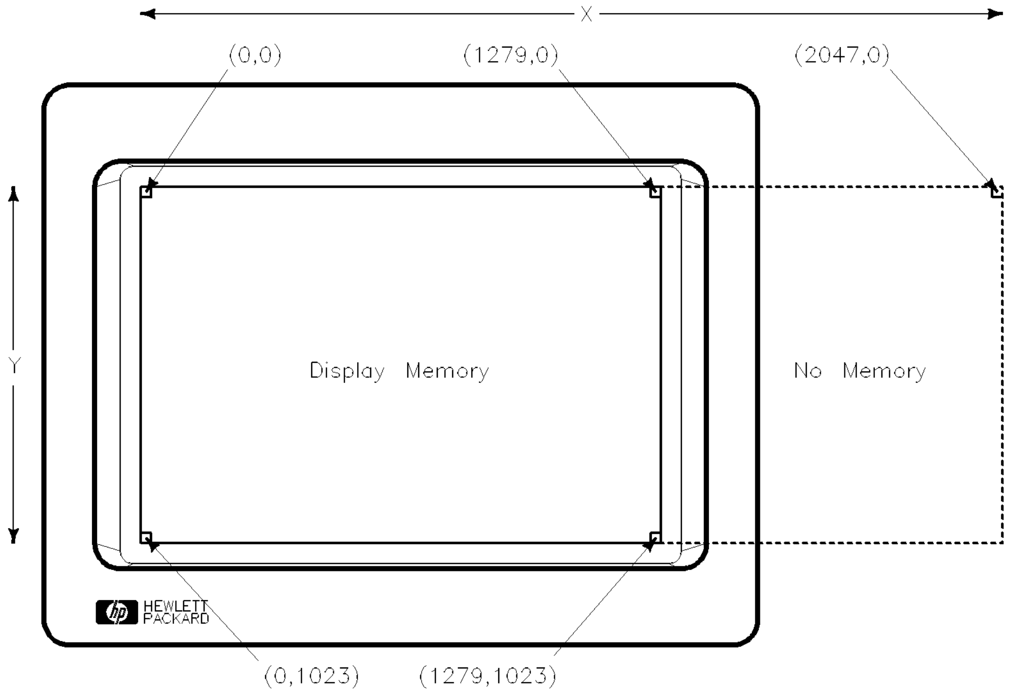

Figure 2-1. Physical Address Space

To Access the Frame Buffer Directly

When using the R_GET_FRAME_BUFFER gescape for direct user access to the frame buffer, correct access can only be assured by using the R_LOCK_DEVICE and R_UNLOCK_DEVICE gescapes.

1. Use R_LOCK_DEVICE just prior to direct frame buffer access.
2. Use R_UNLOCK_DEVICE directly after the frame buffer access and before any other Starbase commands.

Caution Do not read from or write to the offscreen addresses. Such operations will cause undefined behavior.

Frame Buffer Address Mapping

The frame buffer is organized as a single one-dimensional array of pixel values. The first byte (byte 0) of the frame buffer represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1279 is the last (right-most) displayable pixel on the top line. The next 768 bytes are not displayable. Byte 2048 is the first (left-most) pixel on the second line from the top. The last (lower right corner) pixel on the screen is byte number 2,096,383 ($1023 \times 2048 + 1279$).

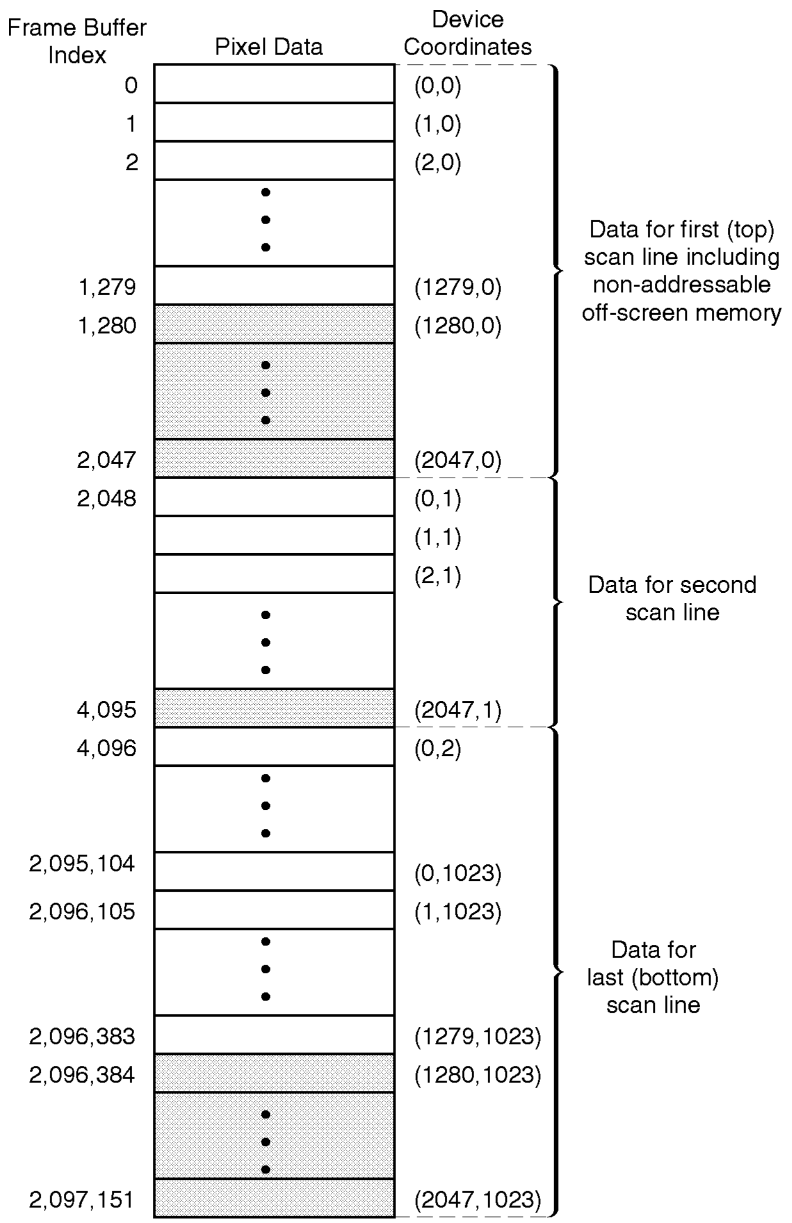

Figure 2-2. Frame Buffer Mapping in Memory

The frame buffer organization is essentially the same for all of the HCRX family of graphics devices, except for the number of banks. The HP VISUALIZE-EG has one bank of 8 planes. The HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 have two banks of 8 image planes (one for each buffer), and the HCRX-24, HCRX-24Z, and HP VISUALIZE-24 have three banks of 8 image planes (one for each color). Only one bank can be accessed at a time.

For `block_read` and `block_write` operations to the image planes, the data is in all eight bits of each byte. The default for reading the Z-buffer is always 24-bits per pixel in a 32-bit word. The Z-buffers for the HCRX-8Z HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 are 23 bits deep, and their Z-buffer data is left justified in the lower 24 bits of the 32-bit word (that is, the 23-bit Z-buffer data is shifted left one bit from the least-significant bit), as shown in the following figure.

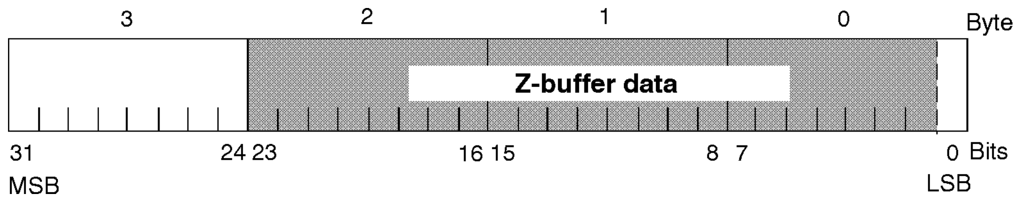

Figure 2-3. Hardware Z-Buffer Data Alignment

For the HP VISUALIZE-EG, HCRX-8 and HCRX-24 with PowerShade, the Z-buffer is 16-bits deep, and their Z-buffer data is left justified in the 32-bit word. The `raw` parameter to `block_read` and `block_write` must be set to true in order to read from or write to the Z-buffer. Using `wbank=3` in the `bank_switch` command on the HCRX-24, HCRX-24Z, and HP VISUALIZE-24, or `wbank=2` on the HP VISUALIZE-EG, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 selects the Z-buffer for reads or writes.

Unlike the frame buffer, the Z-buffer data is contiguous. The HP VISUALIZE-EG, HCRX-8, and HCRX-24 have software Z-buffers, which are the size of the window. For example, if the window is 400x400, word 400 is the leftmost Z-buffer value for the second scan line. The HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 have hardware Z-buffers, which are always 1280x1024 where word 1280 is the leftmost word of the second scanline.

Frame Buffer Configurations

The following table shows which color map modes are supported for different frame buffer configurations. No entry (i.e. blank) indicates no support.

Table 2-2. Supported Frame Buffer Configurations

Number of Planes	HP VISUALIZE-EG, **HCRX-8**, **HCRX-8Z**, **and** HP VISUALIZE-8	**HCRX-24**, **HCRX-24Z**, **and** HP VISUALIZE-24

8, 8/8	`CMAP_NORMAL`
	`CMAP_FULL`
	`CMAP_MONOTONIC`
		`CMAP_NORMAL`
		`CMAP_FULL`
		`CMAP_MONOTONIC`

12, 12/12		`CMAP_FULL`
24		`CMAP_FULL`

Since Starbase supports double-buffering per window, it is better to request double-buffering with a depth of eight on an HP VISUALIZE-EG, HCRX-8, HCRX-8Z, or HP VISUALIZE-8, or a depth of 12 when in `CMAP_FULL` mode on an HCRX-24, HCRX-24Z, or HP VISUALIZE-24.

Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported in depth 8 windows for compatibility with previous devices, however, it is not recommended. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes. Flashing may occur; however, as this kind of double-buffering is not performed on a per window basis.

Device Support for the TrueColor Visual

TrueColor Visual Description

A TrueColor visual can be thought of as having a read-only color map where, for any given pixel value, about one third of the bits are used to describe each of the red, green, and blue colors, respectively. For an 8-plane TrueColor visual, 3 bits describe the red component, 3 bits describe the green component, and 2 bits describe the blue component. A 12-plane TrueColor visual uses 4 bits each to describe the red, green, and blue components. A 24-plane TrueColor visual uses 8 bits each to describe the red, green, and blue components. This is illustrated as follows:

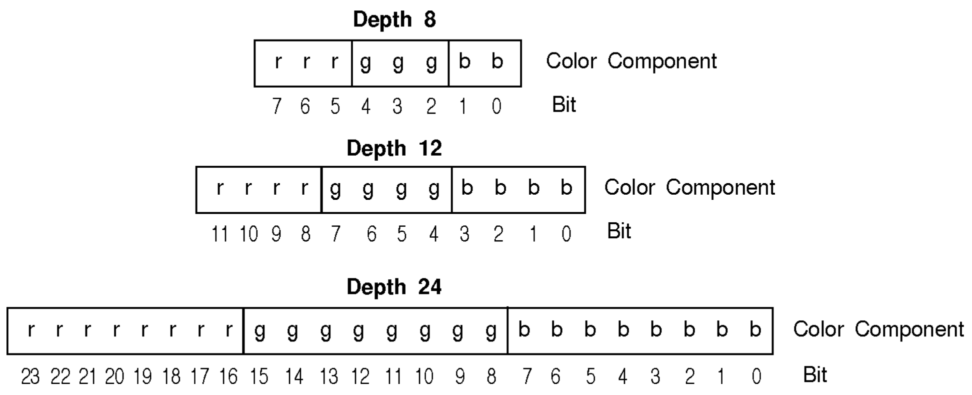

Figure 2-4. Pixel Representation for the 8-, 12-, and 24-Plane TrueColor Visuals

The following example refers to an 8-plane TrueColor visual; however, the example can be expanded to apply to 12-plane or 24-plane TrueColor visuals.

Example

Since the red and green components consist of 3 bits each, there are 8 different shades of red and 8 different shades of green available. Also, there are 4 different shades of blue represented by 2 bits. As the component increases, the intensity of that color increases. For example, a red component of 000 represents no red, and a red component of 111 represents full red. Therefore, pixel value 0 is 000 red, 000 green, and 00 blue, which results in black. Pixel value 255 is 111 red, 111 green, and 11 blue, which results in white. These and other examples are shown in Table 2-3.

Table 2-3. Examples of Pixel Color Values

Pixel Value	Binary	Red	Green	Blue
0	000 000 00	shade 0	shade 0	shade 0
53	001 101 01	shade 1	shade 5	shade 1
139	100 010 11	shade 4	shade 2	shade 3
218	110 110 10	shade 6	shade 6	shade 2
255	111 111 11	shade 7	shade 7	shade 3

Note that the red, green and blue intensities for the color shades ramp uniformly between 0 and 255.

Device Specific Visuals Information

Note that the TrueColor Visual always uses a `shade_mode` of `CMAP_FULL`. When the TrueColor visual window is `gopen`ed, your application will automatically be in `CMAP_FULL` mode and the `shade_mode` call will ignore any attempts to go into another mode.

With the addition of the TrueColor visual (beginning with the 9.03 release of HP-UX), you need to consider the following information:

- If you originally created your windows using command lines similar to the following:

  ```
  xwcreate -g 600x500+300+200 -depth 8 window
  xwcreate -g 600x500+300+200 -depth 24 window
  ```

  you will have to change how you create your windows by using command lines similar to these:

  ```
  xwcreate -g 600x500+300+200 -depth 8 -visual TrueColor window
  xwcreate -g 600x500+300+200 -depth 24 -visual TrueColor window
  ```

  Note the addition of the command line option `-visual` for declaring TrueColor and DirectColor visuals.

- The TrueColor color map is a read-only color map and cannot be modified. You should note that any attempt to modify the TrueColor color map will not cause an error.

  One class of applications that could be affected by this color map description are those that perform their own gamma correction. This is because the color map is read-only, thus preventing Starbase from adjusting the color ramp to perform gamma correction.

- If your application searches for a visual by traversing the visual list that is returned by the X server, you will find that the order of visuals in this list may change from device to device and release to release. Therefore, your application code should always explicitly search for a particular visual rather than assuming that it occurs in a fixed position within the list of visuals returned by X11.

Using Starbase in X Windows

This section contains device specific information needed to run Starbase programs in X11 windows. If you need a general, device-independent explanation of using Starbase in X11 windows, refer to the "Using Starbase with the X Window System" chapter of *Starbase Graphics Techniques*.

To reduce the complexity of having multiple X server modes, the `hphcrx` drivers for X and Starbase only support one X server mode for each device. Several other key features have been designed to improve the overall usability of the devices in the X11 windows environment, and to reduce interaction issues between the X11 user interface and graphics library APIs (such as Starbase) that provide *direct hardware access* (DHA).

Per-Window Double-Buffering

The HCRX family of graphics devices support double-buffering in the images planes on a per-window basis. Note that the HP VISUALIZE-EG supports double-buffering in the images planes on a per-window basis only if it has the add-on memory card installed, or if you use VM double-buffering. The HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 graphics devices support 8/8 double-buffering for each of the Starbase color map modes (`CMAP_NORMAL`, `CMAP_FULL`, `CMAP_MONOTONIC`) in the image planes. In the image planes, the HCRX-24, HCRX-24Z, and HP VISUALIZE-24 support the above mentioned modes, plus 12/12 double-buffered and 24 planes single-buffered in the `CMAP_FULL` color map mode. Any X11 library drawing routines will render to the currently visible buffer of a window that has double-buffering enabled.

Note that Starbase uses the `hpvmx` device driver to perform double-buffering in software in the overlay planes. This double-buffering method is slower than the hardware double-buffering used in the image planes. Any X11 library drawing routines will render to the currently visible buffer of a window that has double-buffering enabled.

Available Color Map Entries

The HCRX family of graphics devices has two hardware color maps in the overlay planes (where overlay planes are supported) and at least two hardware color maps in the image planes.

Using the default X server mode of the HP VISUALIZE-EG, HCRX-8, HCRX-8Z, and HP VISUALIZE-8, if you query the X server for the number of entries in the default overlay visual's color map, the server will reply that there are 256 entries available.

Starbase Color Maps and X11 Read/Write Restrictions

The X color model defines read/write restrictions both on color maps and on individual entries in color maps. As of HP-UX 9.05, Starbase no longer overwrites read-only color maps or color map entries as defined in the X color model. Attempts to write to color map entries in read-only color maps (color maps in StaticGray, StaticColor or TrueColor visuals) are silently ignored.

Accessing HP Color Recovery with X Windows

The HCRX family of graphics devices support HP Color Recovery for shaded areas. When a depth 8 image window is used, HP Color Recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. HP Color Recovery is available on all depth 8 windows on the HP VISUALIZE-EG, HP VISUALIZE-8, HP VISUALIZE-24, HCRX-8, HCRX-8Z, HCRX-24, and HCRX-24Z. For more information about HP Color Recovery, read the section "HP Color Recovery Technology" found in this chapter.

The Starbase, HP PEX, and HP-PHIGS graphics libraries provide programmers who use these APIs with transparent access to the HP Color Recovery capability of the HCRX family of graphics devices. If you are producing graphics using Xlib calls, then your application must perform some of the necessary processing. At server start-up, there is one property that is defined and placed on the root window if the `HP_DISABLE_COLOR_RECOVERY` environment variable has *not* been exported. This property is:

```
_HP_RGB_SMOOTH_MAP_LIST
```

The above property is of type `RGB_COLOR_MAP` and carries pointers to structures of type `XStandardColormap`. It may be interrogated with calls to `XGetRGBColormaps`. The property `_HP_RGB_SMOOTH_MAP_LIST` is a list of color maps that are associated with window visual IDs that support HP Color Recovery. When the `XGetRGBColormaps` routine searches throughout this list for a color map with a visual ID that matches your window's visual ID and it finds such a visual, your application knows that your visual supports HP Color Recovery, and uses that color map for any HP Color Recovery window.

HP Color Recovery uses all 256 entries of one of the available color maps. The color visual used by HP Color Recovery emulates the depth 24 TrueColor visual, thus, the colors red, green, and blue are typically declared as integers in the range from 0 to 255. Note that each window that uses HP Color Recovery will use the same color map.

For HP Color Recovery to produce the best results, the emulated depth 24 TrueColor data is dithered as explained below.

A pixel to be dithered is sent to the routine provided in this example. Note that the values of the variables `RedValue`, `GreenValue` and `BlueValue` are generated by an application. In this example, the color values are assumed to be in the range [0..255].

The given routine receives the color values and the X and Y window address (`Xp` and `Yp`) of the pixel. The X and Y address is used to access the dither tables. The values from the dither tables are added to the color values. After the dither addition, the resultant color values are quantized to 3 bits of red and green and 2 bits of blue. The quantized results are packed into an 8-bit `unsigned char` and then stored in the frame buffer. As the contents of the frame buffer are scanned to the CRT, a special section in the HCRX hardware then converts the 8-bit data stored in the frame buffer into a 24-bit TrueColor image for display.

Here is a routine that can be used to dither the 24-bit TrueColor data.

```
unsigned char dither_pixel_for_CR(RedValue,GreenValue,BlueValue,Xp,Yp)
int     RedValue,GreenValueBlueValue,Xp,Yp;
{
static short dither_red[2][16] = {
  {-16,  4, -1, 11,-14,  6, -3,  9,-15,  5, -2, 10,-13,  7, -4,  8},
  { 15, -5,  0,-12, 13, -7,  2,-10, 14, -6,  1,-11, 12, -8,  3, -9} };

static short dither_green[2][16] = {
  { 11,-15,  7, -3,  8,-14,  4, -2, 10,-16,  6, -4,  9,-13,  5, -1},
  {-12, 14, -8,  2, -9, 13, -5,  1,-11, 15, -7,  3,-10, 12, -6,  0} };

static short dither_blue[2][16] = {
  { -3,  9,-13,  7, -1, 11,-15,  5, -4,  8,-14,  6, -2, 10,-16,  4},
  {  2,-10, 12, -8,  0,-12, 14, -6,  3, -9, 13, -7,  1,-11, 15, -5} };

int red, green, blue;
int x_dither_table, y_dither_table;
unsigned char pixel;

x_dither_table = Xp % 16;   /* X Pixel Address MOD 16 */
y_dither_table = Yp % 2;    /* Y Pixel Address MOD 2 */

red = RedValue;
green = GreenValue;
blue = BlueValue;

if (red >= 48) /* 48 is a constant required by this routine */
   red=red-16;
else
   red=red/2+8;
red += dither_red[y_dither_table][x_dither_table];
if (red > 0xff) red = 0xff;
if (red < 0x00) red = 0x00;

if (green >= 48) /* 48 is a constant required by this routine */
   green=green-16;
else
   green=green/2+8;
green += dither_green[y_dither_table][x_dither_table];
if (green > 0xff) green = 0xff;
if (green < 0x00) green = 0x00;

if (blue >= 112) /* 112 is a constant required by this routine */
   blue=blue-32;
else
   blue=blue/2+24;
blue += (dither_blue[y_dither_table][x_dither_table]<<1);
if (blue > 0xff) blue = 0xff;
if (blue < 0x00) blue = 0x00;

pixel = ((red & 0xE0) | ((green & 0xE0) >> 3) | ((blue & 0xC0) >> 6));

return(pixel);
}
```

Backing Store

The HP VISUALIZE-EG, HCRX-8 and HCRX-24 support backing store (also known as "retained raster"). The backing store feature allows a window being rendered to by a direct hardware access (DHA) client to be "backed-up" to a virtual frame buffer whenever any portion of the window is obscured by another window. In this case, the application is not required to catch "expose events" from X11 and redraw the picture when occlusion occurs. In fact, no expose events will be generated if backing store is enabled.

Thus, when a window is placed on top of another window containing a complete image, the window system will save the contents of the latter window before displaying the obscuring window. Then, when the obscuring window is removed, the earlier contents of the occluded area plus any new rendering that has occurred in the occluded area during the cover-up will be restored. Since rendering to the virtual frame buffer is not as fast as rendering to the actual frame buffer in the occluded area, the performance will suffer, but only while the window is occluded.

Backing Store Exceptions

In general, those Starbase operations that draw to the display are also supported when drawing to backing store. The exceptions to this are:

- Backing store with 24-plane visuals is not supported.
- Backing store cannot be enabled for the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, or HP VISUALIZE-24 accelerator.
- Backing store will not work with certain `gescape` operations that access device-dependent features.
- Backing store contents may be incorrect if you mix Xlib rendering with Starbase rendering to an 8/8 double-buffered window.

If these limitations on support of backing store prove to be troublesome in your application, do not use backing store. Instead, detect window exposure events and repaint the window when a previously obscured portion of a window is made visible.

X11 Cursor

The X11 cursor (often called the sprite) never interferes with the frame buffer contents in either the image or overlay planes.

Supported Visuals

The following table of Supported Visuals contains information for programmers using either Xlib graphics or Starbase. The table lists the image plane depths of windows and color map access modes that are supported for a given graphics device. It also indicates whether or not backing store (retained raster) is available for a given visual, and lists the double-buffer configurations supported by Starbase for this device driver.

Table 2-4. Supported Visuals

Device	Depth	Visual Class	Backing Store		Starbase Double-Buffer[1]
			Xlib	Starbase[2]	
HP VISUALIZE-EG	8	PseudoColor	Yes	Yes	8/8[3]
		TrueColor	Yes	Yes	8/8[3]
HCRX-8, HP VISUALIZE-EG with add-on memory	8	PseudoColor	Yes[4]	Yes	8/8
		TrueColor	Yes[4]	Yes	8/8
HCRX-8Z, HP VISUALIZE-8	8	PseudoColor	Yes[4]	No	8/8
		TrueColor	Yes[4]	No	8/8
HCRX-24	8	PseudoColor	Yes[4]	Yes	8/8
		TrueColor	Yes[4]	Yes	8/8
	12	DirectColor	Yes	No	12/12
		TrueColor	Yes	No	12/12
	24	DirectColor	Yes	No	12/12
		TrueColor	Yes	No	12/12

1 Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported for compatibility with previous devices, however, it is not recommended. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes. Flashing may occur, however, as this kind of double-buffering cannot be done on a per window basis. Note that double-buffering with less than 8-planes is *only* supported in `CMAP_NORMAL`.

2 The `hphcrx` device driver does not support backing store with the `ACCELERATED` *mode* of `gopen`.

3 8-bit double-buffering is done via Virtual Memory (VM) double-buffering. For information on VM double-buffering, read the section "VM Rendering Utilities" found in the chapter "HP Virtual Memory and X" of this manual.

4 Full support for single-buffered windows. The X11 server will not maintain backing store for an obscured Starbase window if double-buffering is turned on and Xlib calls are made to that window. Whenever backing store is not maintained, normal expose events are generated. Also, rendering using an attached accelerator (for example, the HCRX-24Z) does not support backing store.

Table 2-4. Supported Visuals (continued)

Device	Depth	Visual Class	Backing Store		Starbase Double-Buffer[1]
			Xlib	Starbase[2]	
HCRX-24Z, HP VISUALIZE-24	8	PseudoColor	Yes[1]	No	8/8
		TrueColor	Yes[1]	No	8/8
	12	DirectColor	Yes	No	12/12
		TrueColor	Yes	No	12/12
	24	DirectColor	Yes	No	12/12
		TrueColor	Yes	No	12/12

1 Full support for single-buffered windows. The X11 server will not maintain backing store for an obscured Starbase window if double-buffering is turned on and Xlib calls are made to that window. Whenever backing store is not maintained, normal expose events are generated. Also, rendering using an attached accelerator (for example, the HCRX-24Z) does not support backing store.

Overlay Plane Transparency and the X Windows System

An overlay visual's transparency feature enables you to render opaque objects (for example, menus and text) to a transparent overlay window and at the same time view rendered objects in an image window. For example, you may want to show a map of a country without all of its internal borders, and then add the internal borders as you need them. This can be done by creating two X windows: one in the overlay planes and one in the images planes. The country's terrain and boundaries would be drawn in the image planes window and the internal borders in a transparent overlay window.

The default X11 mode on the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 does not provide an overlay visual with a transparent property. There is a special mode that provides overlay transparency on the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 graphics devices. But, by choosing to use overlay planes transparency on the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8, you will limit the number of hardware color maps available.

Note that the HP VISUALIZE-EG with add-on memory only supports overlay transparency in this special mode when it is configured in 8/8 mode. You must configure your system to recognize the add-on HP VISUALIZE-EG memory. This can be set during a system `reboot`.

The HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 Frame Buffer Configuration

Table 2-5 shows the default HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 frame buffer configuration.

Table 2-5.
The Default HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 Frame Buffer Configuration

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	2	no	PseudoColor
image	8	single or double	2	N.A.	PseudoColor TrueColor

To enable transparency, add the following lines to your ⟨*x11-admin*⟩[2]`/X*screens` file before starting HP CDE, HP VUE, or the X11 server.

```
ScreenOptions
   HP_ENABLE_OVERLAY_TRANSPARENCY
```

or the preferred entry:

```
ScreenOptions
   EnableOverlayTransparency
```

[2] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

This screen option can be disabled by removing it from your `X*screens` file and restarting HP CDE, HP VUE, or the X11 server.

Note The screen option `EnableOverlayTransparency` only has meaning when used with the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 graphics devices. It is ignored by all other graphics devices.

Table 2-6. HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 Frame Buffer Configuration in Overlay Transparent Mode

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	1	yes	PseudoColor
image	8	single or double	1	N.A.	PseudoColor TrueColor

Note that on the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 graphics devices, the use of overlay transparency limits your system to one hardware color map in the overlay planes and one in the image planes.

Using the overlay visual (by default or explicitly) will provide access to a transparent color map entry. However, because the number of hardware color maps is halved in this mode, there is a higher likelihood that you will experience the *technicolor effect*, especially in overlay windows, which typically make up the user interface.

The number of overlay color map entries is now 255 because the last entry is the transparent color map value. If your application requires a report of 256 entries in your color map, you can enable the screen option:

```
ScreenOptions
   CountTransparentInOverlayVisual
```

in the ⟨*x11-admin*⟩[3]`/X*screens` file before starting HP CDE, HP VUE, or the X11 server.

The HCRX-24, HCRX-24Z, and HP VISUALIZE-24 Frame Buffer Configuration

Table 2-7 shows the default HCRX-24, HCRX-24Z, and HP VISUALIZE-24 frame buffer configuration. This configuration is not changed by using overlay transparency.

Table 2-7. The Default HCRX-24, HCRX-24Z, and HP VISUALIZE-24 Frame Buffer Configuration

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	2	no	PseudoColor[1]
overlay	8	single	2	yes	PseudoColor
image	8	single or double	2	N.A.	PseudoColor[2] TrueColor
image	12	single or double	2	N.A.	DirectColor TrueColor
image	24	single	2	N.A.	DirectColor TrueColor

1 This is the default color map mode.

2 This is the first visual returned by `xdpyinfo`.

[3] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

If you need an overlay color map that supports transparency, create the color map using the visual that has transparency in its SERVER_OVERLAY_VISUALS property. To look at the contents of this property, you would use code similar to the following:

```
/* First, get the list of visuals for this screen. */
    .
    .
*pVisuals = XGetVisualInfo(display, mask, &getVisInfo, numVisuals);
    .
    .
/* Now, get the overlay visual information for this screen.  To obtain
 * this information, get the SERVER_OVERLAY_VISUALS property. */

overlayVisualsAtom = XInternAtom(display, "SERVER_OVERLAY_VISUALS", True);
if (overlayVisualsAtom != None)
{
    /* Since the Atom exists, we can request the property's contents. */
    bytesAfter = 0;
    numLongs = sizeof(OverlayVisualPropertyRec) / 4;
    do
    {
        numLongs += bytesAfter * 4;
        XGetWindowProperty(display, RootWindow(display, screen),
                           overlayVisualsAtom, 0, numLongs, False,
                           overlayVisualsAtom, &actualType, &actualFormat,
                           &numLongs, &bytesAfter, pOverlayVisuals);
    } while (bytesAfter > 0);
}
    .
    .
/* Process the pOverlayVisuals array. */
while (--nVisuals >= 0) {
    nOVisuals = *numOverlayVisuals;
    pOVis = *pOverlayVisuals;
    imageVisual = True;
    while (--nOVisuals >= 0) {
        pOOldVis = (OverlayVisualPropertyRec *) pOVis;
        if (pVis->visualid == pOOldVis->visualID)
        {
            imageVisual = False;
            pOVis->pOverlayVisualInfo = pVis;
            /* Found the transparent visual */
            if (pOVis->transparentType == TransparentPixel);
        }
        pOVis++;
    }
}
```

This program segment is not complete; however, its main purpose is to give you an idea of how a visual is checked for overlay transparency. The source for the above code can be found in the file[4]:

⟨*sb-utils*⟩`/wsutils.c`

HCRX Configuration Hints

This section discusses:

- Visuals and double-buffer support for HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8.
- Moving the default visual to the image planes.

[4] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 Visuals and Double-Buffer Support

The HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 have 8 overlay and 16 image planes. Note that the 16 image planes can be used for 8/8 double-buffered or 8-planes single-buffered configurations; 16 planes single-buffered is not supported. The following list contains information about the HP VISUALIZE-EG with add-on memory, HCRX-8, HCRX-8Z, and HP VISUALIZE-8, the default visual, and support for double-buffering:

- There are two depth 8 PseudoColor visuals: one in the overlay planes and one in the image planes, as well as a depth 8 TrueColor visual in the image planes.
- The default visual (where the root window and default color map reside) is in the overlay planes. The default visual can be moved to the image planes by making a change in the `X*screens` file as explained in the subsequent section "Moving the Default Visual to the Image Planes."
- Fast 8/8 double-buffering (two hardware buffers) is supported in the depth 8 image planes but not in the depth 8 overlay planes. The overlay planes support the slower memory-based double-buffering.

Programming Recommendations for the Default Visual

Here are some programming suggestions that will help you when working with the default visual.

- The default visual's color map cannot be used with a window in the non-default visual, even one of the same depth.

  Programming Recommendation: Before creating a depth 8 window, first check that the window is in the default visual before trying to use the default color map. If the window is not in the default visual, you must create a color map in that visual. This is exactly the same process that you must follow to create windows in a depth 12 or depth 24 visual.

  Workaround: If you have an application that assumes the default color map can be used with any depth 8 window, even one in the image planes, move the default visual to the image planes as described in the subsequent section "Moving the Default Visual to the Image Planes."

- Unlike the CRX's default visual, the HP VISUALIZE-EG with add-on memory, HP VISUALIZE-8's, HCRX-8's, and HCRX-8Z's default visuals do not have fast hardware double-buffering, but the image planes do.

  Programming Recommendation: To obtain hardware double-buffering, find a visual in the image planes. The best way to do this is to find all the depth 8 PseudoColor visuals returned by `XGetVisualInfo`, then subtract the visuals that are reported in the `SERVER_OVERLAY_VISUALS` property.

  Workaround: If you have an application that assumes the default visual has fast double-buffering, move the default visual to the image planes as explained in the subsequent section "Moving the Default Visual to the Image Planes."

Moving the Default Visual to the Image Planes

Note By default, overlay planes have the default X11 color map permanently locked into one hardware color map, and the second hardware color map is available for applications to use. Moving the default visual into the image planes will limit the number of hardware color maps available to you.

In this mode, the HCRX family of graphics devices provide a single hardware color map in the overlay planes. This hardware color map will be shared by X11 and the applications currently being executed. The application (or X11) whose color map is currently downloaded into hardware will look correct; the other applications or X11 may experience the *technicolor effect*.

X Windows provides a method for changing the default visual from a depth 8 overlay PseudoColor visual to a depth 8 image PseudoColor visual. To do this, use SAM. Or, you can manually edit the file[5]:

⟨*x11-admin*⟩`/X*screens`

and add the following lines:

```
Screen ⟨dev⟩/crt
   DefaultVisual
      Class PseudoColor
      Depth 8
      Layer Image
```

The `*` in the `X*screens` file name specifies the *display_number*. To determine the display number, execute this shell command:

```
echo $DISPLAY
```

Your results will have the following syntax:

⟨*host_name*⟩`:`⟨*display_number*⟩`.`⟨*screen_number*⟩

[5] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Here is an example of what your display name might look like after executing the `echo $DISPLAY` shell command:

```
mysystem:0.0
```

where *host_name* is `mysystem`, *display_number* is 0, and *screen_number* is 0. In the above example, you would edit the file:

⟨*x11-admin*⟩`/X0screens`

Note that the syntax of this specification has changed. For more information, see the file:

⟨*x11*⟩`/Xserver/info/screens/hp`

The Overlay Plane Color Map Management Scheme

Many applications use the default X11 color map. A technicolor (color flashing) effect in the windows using the default color map occurs when a non-default color map is downloaded into the hardware color map that had previously contained the default X11 color map.

Because so many applications use the default X11 color map, and because the HCRX family of graphics devices have two hardware color maps in the overlay planes, the behavior on these devices is to dedicate (that is, lock) one overlay hardware color map to always hold the default X11 color map. This means that the assigned default overlay hardware color map cannot have another color map downloaded to it. The other overlay hardware color map is available to applications that use color maps other than the default.

Starbase Echoes

This section provides information about the echo implementation for Starbase. The HCRX family of graphics devices and all future graphics device drivers will use this implementation of Starbase echoes.

Starbase echoes use Xlib functionality to draw echoes in the same planes as the visual that is active for the window. All previously supported Starbase echo functions are implemented except for those listed below:

- There is no support for the following gescapes:

  ```
  R_DEF_ECHO_TRANS
  R_ECHO_FG_BG_COLORS
  R_OV_ECHO_COLORS
  R_OVERLAY_ECHO
  ```

  In addition, the `R_ECHO_MASK` is only supported for a maximum of two colors within the raster definition.
- More than one Starbase echo per window is not supported.
- Rendering with both Xlib and Starbase in the same window while Starbase echoes are active may produce some random pixel “noise”.

To Open and Initialize the Device for Output

Syntax Examples

The following examples show how to open the graphics devices for output[6]:

C programs:

```
fildes = gopen("⟨screen⟩/window", OUTDEV, NULL, INIT);
```

FORTRAN77 programs:

```
  fildes = gopen('⟨screen⟩/window'//char(0), OUTDEV,
+ char(0), INIT)
```

[6] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Pascal programs:

```
fildes := gopen('⟨screen⟩/window', OUTDEV, '', INIT);
```

Parameters for gopen

The `gopen` procedure has four parameters:*path*, *kind*, *driver*, and *mode*.

- *path* — This is the name of the device file created by `xwcreate`(1) or created with `XCreateWindow`(3X) and returned from `make_X11_gopen_string`(3G).
- *kind* — This parameter should be `OUTDEV` if the window will be used for output, `INDEV` if the window will be be used for Starbase input, or `OUTINDEV` if the window will be used for both output and Starbase input.
- *driver* — The character representation of the driver type. If this parameter is set to `NULL`, then `gopen` will inquire the device and use the appropriate driver. Where there are both accelerated and unaccelerated versions of the driver, the default is to load the accelerated version.

  For example:

`NULL`	*for C.*
`char(0)`	*for FORTRAN 77.*
`''`	*for Pascal.*

  Or, a character string may be used to specify a driver. For example:

`"hphcrx"`	*for C.*
`'hphcrx'//char(0)`	*for FORTRAN 77.*
`'hphcrx'`	*for Pascal.*

- *mode* — The mode control word consists of several flag bits *OR*ed together. Listed below are flag bits that have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure. See the *Starbase Graphics Techniques* manual for a description of `gopen` actions when accessing an X11 Window.
  - `0` (zero) — Open the device, but do nothing else. The software color table is initialized from the current state of the hardware color map.
  - `INIT` — Open and initialize the device as follows:
    1. Window is cleared to 0s.
    2. The color map is reset to its default values.
    3. The display is enabled for reading and writing.
    4. Clear the Z-buffer (on the HP VISUALIZE-EG, HCRX-8 or HCRX-24 with PowerShade or on the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, or HP VISUALIZE-24)
  - `RESET_DEVICE` — Same as INIT.
  - `SPOOLED` — Not supported; raster devices cannot be spooled.
  - `MODEL_XFORM` — Opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
  - `INT_XFORM` — Perform only integer and common operations. All floating point operations will cause an error.
  - `INT_XFORM_32` — Perform only integer and common operations, with extended precision. All floating point operations will cause an error.

    This mode is provided for compatibility of integer precision with previous devices. `INT_XFORM` might use a faster transformation pipeline with slightly less precision. It is recommended to use `INT_XFORM` unless maximum precision is required. If maximum precision is required, even at the expense of performance, use `INT_XFORM_32`.
  - `FLOAT_XFORM` — Perform only floating point and common operations. All integer operations will cause an error.
  - `UNACCELERATED` — Tells the `hphcrx` driver to not use the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, or HP VISUALIZE-24 accelerator. This flag only applies when `NULL` is specified for the driver name.
  - `ACCELERATED` — Tells the `hphcrx` driver to use the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, or HP VISUALIZE-24 accelerator if present (default). This flag only applies when `NULL` is specified for the driver name.

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location (0, 0) is the upper-left corner of the window with X-axis values increasing to the right and Y-axis values increasing down.

Use this form of pixel addressing when calling high-level Starbase operations in terms of (`X,Y`) device coordinates.

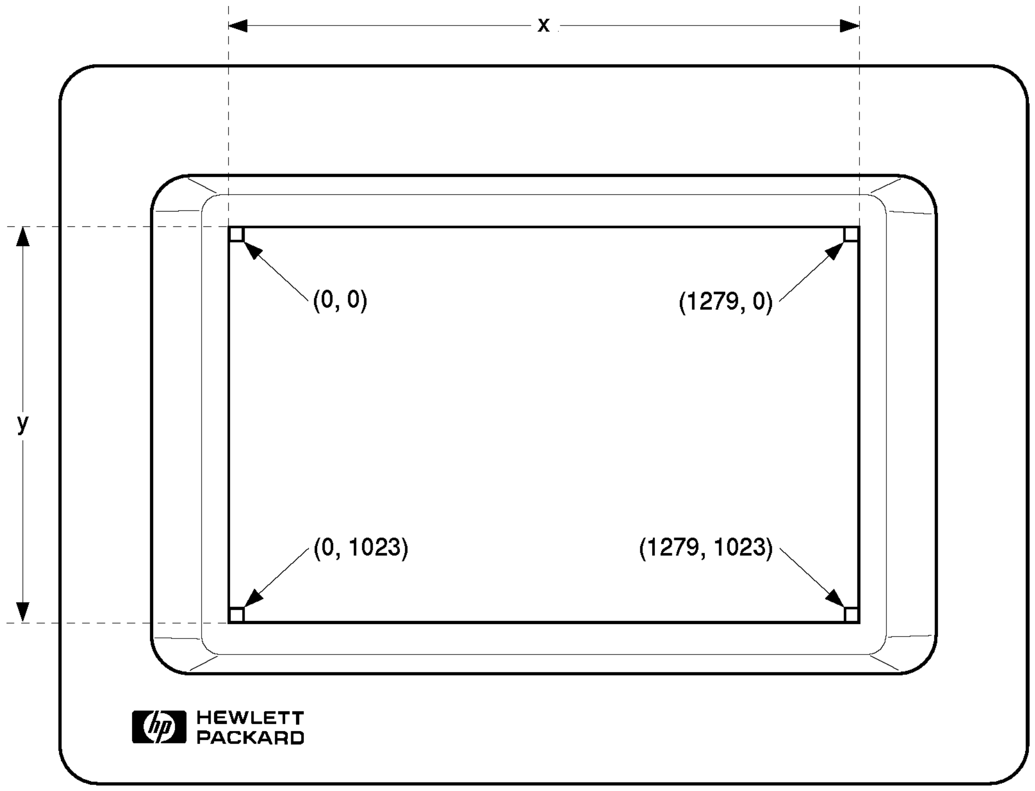

Figure 2-5. Device Coordinates

Starbase Functionality

This section contains information on Starbase calls that are *not* supported by the HCRX family of graphics devices and on `gescape`s that are supported by the HCRX family of graphics devices.

Note that for texture mapping, the older, texture_*, calls are not supported on these devices. But, the new, tm_*, calls are supported on all HCRX devices.

Calls Not Supported on the HCRX-8Z and HP VISUALIZE-8, or on the HP VISUALIZE-EG and HCRX-8 with PowerShade

The following calls are not supported when using the HP VISUALIZE-8 and HCRX-8Z graphics devices, or when using an HP VISUALIZE-EG or HCRX-8 graphics device with PowerShade 3D Surfaces Software.

```
alpha_transparency
bf_alpha_transparency
bf_texture_index
contour_enable
define_contour_table
define_texture
line_filter
perimeter_filter
texture_index
texture_viewport
texture_window
```

Calls Not Supported on the HCRX-24Z and HP VISUALIZE-24, or on the HCRX-24 with PowerShade

The following calls are not supported when using PowerShade 3D Surfaces Software with the HCRX-24 graphics device, or when using the HCRX-24Z and HP VISUALIZE-24 graphics devices:

```
bf_texture_index
contour_enable
define_contour_table
define_texture
texture_index
texture_viewport
texture_window
```

In addition, `alpha_transparency`, `line_filter`, and `perimeter_filter` are *not* supported on the HCRX-24, with or without PowerShade.

Calls not Supported on the HP VISUALIZE-EG, HCRX-8, and HCRX-24

The hphcrx driver does not support the following Starbase calls on the HP VISUALIZE-EG, HCRX-8, and HCRX-24 graphics devices if you are using Starbase without the PowerShade software. When executed, these calls will produce no result (that is, they are ignored).

```
alpha_transparency              hidden_surface
backface_control                light_ambient
bf_alpha_transparency           light_attenuation
bf_control                      light_model
bf_fill_color                   light_switch
bf_interior_style               line_filter
bf_perimeter_color              perimeter_filter
bf_perimeter_repeat_length      set_capping_planes
bf_perimeter_type               set_model_clip_indicator
bf_surface_coefficients         set_model_clip_volume
bf_surface_model                surface_coefficients
bf_texture_index                surface_model
contour_enable                  texture_index
define_contour_table            texture_viewport
define_texture                  texture_window
define_trimming_curve           viewpoint
deformation_mode                zbuffer_switch
```

Conditional Support of Starbase Calls on the HP VISUALIZE-EG and HCRX-8

The following call is supported with the listed exceptions:

`block_read`, `block_write`	The *raw* parameter for the `block_read` and `block_write` commands is used by this driver to do plane-major reads and writes. It is enabled by the gescape `R_BIT_MODE`.

The storage destination supplied by the user as the source or destination must be organized as follows.

- The data from each plane is packed with eight pixels per byte.
- Each row must begin on a byte boundary. Thus the size of the rectangle as specified by the ⟨*length_x*⟩ and ⟨*length_y*⟩ parameters must correspond to an integral number of bytes.
- The data for the next plane begins on the following byte boundary.
- Clip to the window limits.
- The first pixel in the source rectangle is placed in the high-order bit of the first byte in each plane region.
- When clipping, part of each plane region will not be read (`block_read`) or altered (`block_write`).

A bit mask selects the planes to read or write. The initial value of this mask is 1 (one) indicating that only plane 0 is to be accessed. The value of the mask may be changed using the `R_BIT_MASK` or `GR2D_PLANE_MASK` gescapes. `GR2D_PLANE_MASK` is discussed in the appendix of the *Starbase Device Drivers Manual*. The planes selected by the mask are expected to reside in consecutive plane locations in the user storage area. This reduces the storage requirements to exactly what is needed but also presents the potential for addressing violations or undesirable results.

For example, if the plane mask is changed to specify more planes between a `block_read` and a following `block_write` from the same location, the `block_write` will attempt to access storage for planes that were not read (and perhaps not allocated). The application program must ensure consistency in these operations.

Conditional Support of Starbase Calls on the HCRX-24, HCRX-24Z, or HP VISUALIZE-24

The following calls are supported with the listed exceptions:

`alpha_transparency` The HCRX-24Z and HP VISUALIZE-24 graphics devices support alpha transparency. Alpha only applies to filled areas such as polygons, quadrilateral meshes, triangular strips, and spline surfaces. Vector primitives are not rendered with alpha transparency. The alpha transparency feature is limited to `CMAP_FULL` in the 12/12 or 24-plane configurations. Only the floating point version of these primitives will be rendered with alpha transparency; device coordinate primitives do NOT use alpha. The HCRX-24Z and HP VISUALIZE-24 do not support alpha transparency with attenuation. (See `alpha_transparency`(3G) *Starbase Reference Manual* for a list of parameters).

`block_read`, `block_write` The *raw* parameter for the `block_read` and `block_write` commands is used by this driver to do plane-major reads and writes. It is enabled by the gescape `R_BIT_MODE`.

The storage destination supplied by the user as the source or destination must be organized as follows.

- The data from each plane is packed with eight pixels per byte.
- Each row must begin on a byte boundary. Thus the size of the rectangle as specified by the ⟨*length_x*⟩ and ⟨*length_y*⟩ parameters must correspond to an integral number of bytes.

- The data for the next plane begins on the following byte boundary.
- Clip to the window limits.
- The first pixel in the source rectangle is placed in the high-order bit of the first byte in each plane region.
- When clipping, part of each plane region will not be read (`block_read`) or altered (`block_write`).

A bit mask selects the planes to read or write. The initial value of this mask is 1 (one) indicating that only plane 0 is to be accessed. The value of the mask may be changed using the `R_BIT_MASK` or `GR2D_PLANE_MASK` gescapes. `GR2D_PLANE_MASK` is discussed in the appendix of this manual. The planes selected by the mask are expected to reside in consecutive plane locations in the user storage area. This reduces the storage requirements to exactly what is needed but also presents the potential for addressing violations or undesirable results.

For example, if the plane mask is changed to specify more planes between a `block_read` and a following `block_write` from the same location, the `block_write` will attempt to access storage for planes that were not read (and perhaps not allocated). The application program must ensure consistency in these operations.

`fill_dither`

The ability to dither is disabled if the number of colors specified by `fill_dither` is one. However, if the number of colors specified is greater than 1, the default dither cell size of 16 is used. Dithering is only used in depth 8 windows while in either the `CMAP_FULL` or `CMAP_MONOTONIC` color map mode. Note that the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 devices do not dither when shading in `CMAP_MONOTONIC` mode.

`interior_style`

The *style*s `INT_PATTERN` and `INT_HATCH` are not supported by the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 configurations when the accelerator is being used.

`light_source`

Up to 15 directional light sources are available with the `hphcrx` device driver. The HP VISUALIZE-8 and HP VISUALIZE-24 hardware accelerates up to eight directional light sources. Using nine to fifteen directional light sources will cause a noticeable performance degradation.

`line_filter`, `perimeter_filter`

Anti-aliasing is supported only on the HCRX-24Z and HP VISUALIZE-24. Anti-aliasing for this device applies only to floating point vectors. Device coordinate primitives do not use anti-aliasing. The anti-aliasing features are also limited to the `CMAP_FULL` color map mode in the 12/12 or 24-plane configurations.

The procedures `line_filter` and `perimeter_filter` can be use to specify the two anti-aliasing modes provided by the HCRX-24Z and HP VISUALIZE-24 graphics devices. The index values are assigned as follows:

`0` Anti-aliasing disabled, all vectors have one pixel wide output.

`1` Anti-aliasing enabled, all vectors have two-pixel-wide output. Pixel values are multiplied by the alpha value and blended with the background according the the formula:

$$\mathit{pixel\_color} = (\mathit{new\_pixel} \times \alpha) + (\mathit{old\_pixel} \times (1 - \alpha));$$

`2` Anti-aliasing enabled, all vectors have two-pixel-wide output. Pixel values are multiplied by the alpha value and blended with the background according to the formula:

$$\mathit{pixel\_color} = (\mathit{new\_pixel} \times \alpha) + \mathit{old\_pixel}$$

`pattern_define`	For the HCRX family of graphics devices, the maximum pattern size is 4×4. If a pattern larger than 4×4 is specified, an error message is printed and the previous pattern is retained. The HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 graphics devices do not render pattern fill areas.
`shade_mode`	The color map mode may be selected. Shading can be turned on only if using PowerShade. Shading is not supported on device coordinate primitives even with PowerShade. Note that HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 configurations automatically use PowerShade. Also note that the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 devices do not dither when shading in `CMAP_MONOTONIC` mode.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	Without PowerShade, the ⟨*use*⟩ parameter must be zero. Any extra coordinate information will be ignored. If using PowerShade software, `vertex_format` is fully functional. Note that the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 configurations automatically use PowerShade.
`*_with_data`	The following routines are called `with_data` routines because they allow you to send extra vertex data. These `with_data` routines are supported by the HCRX family of graphics devices.

```
partial_polygon_with_data3d
polygon_with_data3d
polyhedron_with_data
polyline_with_data3d
polymarker_with_data3d
polyquad_with_data3d
polytriangle_with_data3d
quadrilateral_mesh_with_data
triangle_strip_with_data
```

Note that the TEXTURE_MAP flag applies to the TurboVRX devices via the texture_* routines. This extra data per vertex is not used in the tm_* routines.

For detailed information on these routines, read the *Starbase Reference* and "Appendix A" of this document. In some cases, you will be able to find the routines under their own name, but in other cases, you will need to use the first part of the routine name to locate these routines (e.g., polyline_with_data3d is described on the man page for polyline(3G)).

Supported Gescapes

The hphcrx device driver supports the following gescape operations on all HCRX configurations.

- BLOCK_WRITE_SKIPCOUNT—Specify byte skip count during block write.
- COLOR_RECOVERY_CONTROL—Disable HP Color Recovery.
- GCRX_PIXEL_REPLICATE—Pan and zoom a raster image.
- IGNORE_RELEASE—Trigger only when button pressed.
- R_BIT_MASK—Bit mask.
- R_BIT_MODE—Bit mode.
- R_GET_FRAME_BUFFER—Read frame buffer address.
- R_LINE_TYPE—User defined line style and repeat length.
- R_LOCK_DEVICE—Lock device.
- R_READ_FB—Write an image to a window whose shade mode is set to CMAP_FULL.
- R_WRITE_FB—Read the image out of a window created with the shade mode set to CMAP_FULL.
- R_UNLOCK_DEVICE—Unlock device.
- READ_COLOR_MAP—Read Color Map.
- SWITCH_SEMAPHORE—Semaphore Control.
- TRIGGER_ON_RELEASE—Trigger only when button is released.
- STEREO—Activate stereo output mode.

Additional Gescapes for the HCRX-24, HCRX-24Z, and HP VISUALIZE-24

- CUBIC_POLYPOINT—Specify voxels to be rendered in a cubic volume specified in modeling coordinates.
- DC_PIXEL_WRITE—Specify voxels to be rendered along a horizontal scan line.
- GAMMA_CORRECTION—Enable/disable gamma correction.
- LINEAR_POLYPOINT—Specify voxels to be rendered along a line specified in modeling coordinates.

Additional Gescapes for the HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24

- DRAW_POINTS—Select different modes of rounding for rendered points.
- SET_POLYGON_OFFSET—Enable Z-buffer biasing of filled primitives (such as polygons and triangular_strips).

Additional Gescapes Supported with PowerShade on HCRX-8 and HCRX-24, and on HCRX-8Z, HCRX-24Z, HP VISUALIZE-8, and HP VISUALIZE-24 Devices

- `ILLUMINATION_ENABLE`—Turn on/off illumination bits.
- `LS_OVERFLOW_CONTROL`—Set light source overflow handling.
- `POLYGON_TRANSPARENCY`—Segment control over front/back face screen.
- `TRANSPARENCY`—Set screen door transparency mask (front face and back face).
- `WIDELINE_CONTROL`—Turn on/off and set attributes of widelines.
- `ZBANK_ACCESS`—Enable/disable Z-buffer block operations.
- `ZWRITE_ENABLE`—Enable/disable replacement of Z value.

Exceptions to Gescape Support

Note Because the gescape operations are device-dependent, the exceptions discussed below may be removed in future drivers.

`GAMMA_CORRECTION` Gamma correction is implemented by modifying the color map. It is available only in 12-bit or 24-bit DirectColor visuals. For information on the gescape `GAMMA_CORRECTION`, refer to Appendix A in the *Starbase Device Drivers Manual*. If a global gamma correction value has been set via the X server or the gamma correction tool, that global gamma correction value will be used and the color map will not be modified. See the *Graphics Administration Guide* for more information about the gamma correction tool.

When the gescape operations listed below are used with a backing store graphics window, they will have the desired effect for the visible portion of the window, but may cause the backing store for obscured parts to be altered in inconsistent ways. The features involved (along with the names of the affected gescape operations) are listed below. For more details on the gescape operations, refer to Appendix A in the *HP-UX Starbase Device Drivers Manual*.

`R_BIT_MASK` The gescape operation `R_BIT_MASK` defines a plane mask to the driver that is used for bit/pixel access to a single plane in the frame buffer. As with other device drivers, only the plane corresponding to the highest bit set in the mask is transferred.

`R_BIT_MODE` When `block_read` or `block_write` are used with the *raw* parameter set to `TRUE`, the driver supports bit/pixel frame buffer access to single planes.

Porting from HCRX-8 or HP VISUALIZE-EG to HCRX-24 or HP VISUALIZE-24

This section discusses `CMAP_FULL` mode and color plane considerations for porting from an HCRX-8 or HP VISUALIZE-EG to an HCRX-24 or HP VISUALIZE-24.

CMAP_FULL Mode

On an HP VISUALIZE-EG, HCRX-8, HCRX-8Z, and HP VISUALIZE-8, 8 planes will be used in 3:3:2 mode when rendering in the `CMAP_FULL` mode. In 3:3:2 mode, the 8 planes are divided into three planes of red, three planes of green, and two planes of blue. On an HCRX-24, HCRX-24Z, and HP VISUALIZE-24, there are 8 planes for each of the three colors. These differences should not affect your code unless your application needs to perform block operations. The number of colors available will affect your output (the more colors, the better the picture quality). The picture quality looks better on the 24-plane devices.

Although it is possible to do `CMAP_FULL` rendering into 8 planes on an HCRX-24, HCRX-24Z, and HP VISUALIZE-24 the performance will be lower than when using all 24 planes. For higher performance and better image quality, HP recommends using 24 planes.

Number of Color Planes

The HP VISUALIZE-EG, HCRX-8, HCRX-8Z, and HP VISUALIZE-8 have two banks of 8 color planes each. The Starbase `bank_switch` function is used to select bank 0 or 1 for block operations such as `block_read`, `block_write`, and `block_move`. The HCRX-24 and HP VISUALIZE-24 have 24 image planes and 8 overlay planes. The 24 image planes on these devices are organized as three banks of 8 planes each. If an 8-bit window is opened in the image planes, then the HCRX-24, HCRX-24Z, and HP VISUALIZE-24 planes are accessed just like the HCRX-8. If a depth 24 window is opened, then the `bank_switch` function is used to select bank 0, 1, or 2 for block operations. If double buffering is enabled for the depth 24 window, care must be taken to properly format data written to or read from each bank. See the "Block Operations" section earlier in this chapter for information about how to do this.

Porting from HCRX-24 to an HCRX-24Z or HP VISUALIZE-24

The HCRX-24Z and HP VISUALIZE-24 provide accelerators for the HCRX-24 device. The HCRX-24, HCRX-24Z, and HP VISUALIZE-24 use the `hphcrx` device driver.

The HCRX-24Z and HP VISUALIZE-24 accelerated Starbase graphics devices are highly compatible with the HCRX-8 and HCRX-24 graphics devices. This can allow applications written and delivered for HCRX-24 to use the HCRX-24Z and HP VISUALIZE-24 accelerators without requiring different executable code.

Source Incompatibilities

The gescapes available for the HCRX-24Z and HP VISUALIZE-24 are a superset of those available on HCRX-8, HCRX-24, and HP VISUALIZE-8.

Possible behavioral differences between the HCRX-8 and HCRX-24 and HCRX-24Z or HP VISUALIZE-24 should not affect the operation of the application and may only be observed when directly comparing the images between the accelerated and unaccelerated driver. Some of these differences are discussed below.

Backing Store

Backing store is an X11 feature that allocates main memory for obscured regions of a window. Graphics operations are written to this memory as well as the screen. When the window is unobscured, the screen is updated from this memory. This feature is supported by the HCRX-8, HP VISUALIZE-EG, and HCRX-24, but not by the HCRX-24Z and HP VISUALIZE-24. Therefore, applications in the X environment should capture and act on expose events and redraw the image when one is received. X events are documented in the *Programming with Xlib* manual.

"Save under" is a feature of X11 that saves and restores the obscured region of a window when covered by a transient window, such as a menu. If any graphics activity occurs to the obscured window, the save under is voided. This feature is not supported when the transient window is opened with the HCRX-24Z or HP VISUALIZE-24 accelerator.

Note Support for both backing store and save under may change in future releases of the Starbase graphics library.

Image Differences

Because of the different mechanisms used to generate the image when using the HCRX-24Z and HP VISUALIZE-24 accelerator, there may be minor visual differences between accelerated and unaccelerated images. These minor differences are listed below.

- Different visibility properties. Very small primitives may be invisible, that is, where the rendering starts and stops on the same pixel, such as the dot on the letter i. If every pixel is required, see the `DRAW_POINTS` gescape.
- Slight shifts in the image location on the display.
- Minor differences in color interpolation.
- Minor differences in pattern alignment or line type segment alignment.
- Smooth-shaded images in `CMAP_MONOTONIC` colormap mode are never dithered and may appear banded.

Note These differences should be minor. They may change in future releases of the Starbase graphics library.

`screenpr` for the HCRX Family of Graphics Devices

The implementation of *screenpr(1G)* on the HCRX family of graphics devices uses X11 and the HP imaging library, rather than Starbase, to read the screen. This implementation correctly processes image and overlay planes, multiple color maps, and overlay transparency.

Note that command line options are still the same; however, if `screenpr` detects it is running on an HCRX graphics device, it will use the `DISPLAY` environment variable to determine the screen to read, rather than using the device file path given by the `-F` option.

The `-p` option to print a single plane (and consequently the `-f` and `-b` options) is not supported on the HCRX version of `screenpr`. These options will be ignored by `screenpr`.

This version of `screenpr` uses X11, image library calls, and executes *pcltrans(1G)* to produce PCL output. Therefore, `screenpr` may produce error messages from X, the HP Image API library, or `pcltrans`.

The HCRX `screenpr` implementation always expands the data to 24 bits. Therefore, the PCL output of an 8-bit only device like the HCRX family of graphics devices will be approximately three times larger than might be expected.

3 The HP VISUALIZE-48 and HP VISUALIZE-48XP Devices

The `hphcrx48z` driver supports the HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices. These devices provide hardware support for the following operations:

- Generating vectors
- Write-enabling planes and selecting individual banks in the frame buffer
- Writing pixels to the frame buffer with a given replacement rule
- Moving a block of pixels within the frame buffer
- Double-buffering per window
- Flat shaded rectangles
- Dithering and HP Color Recovery technology
- Overlay plane transparency (X Windows)
- Gouraud Shading
- Positional and directional lighting calculations
- Anti-aliased lines
- Alpha transparency
- Hidden surface removal with Z-buffer
- Z-buffering of voxels
- Model geometry transformations
- Optional hardware acceleration of texture mapping

The `hphcrx48z` device driver supports the high performance graphics device described in subsequent sections.

Graphics Device Configuration

Note The HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices do not provide raw-mode graphics support. You must display your Starbase applications in an X11 window or windows. For information using Starbase with X11, read the chapter "Using Starbase with the X Window System" in the *Starbase Graphics Techniques* manual.

The HP VISUALIZE-48 and HP VISUALIZE-48XP support the configuration shown in Table 3-1.

Table 3-1. Supported Depth of Image Plane Windows

Graphics Devices	Number of Image Planes	Number of Overlay Planes	Hardware Accelerator	Resolution
HP VISUALIZE-48, HP VISUALIZE-48XP	8/8, 24/24	8[1]	Yes	1280×1024

1 All Starbase graphics rendering to the overlay planes is done by the VMX or SOX11 device driver.

In order to reduce flickering, this graphics device refreshes the attached CRT displays at least at a 72 Hz rate.

The HP VISUALIZE-48 and HP VISUALIZE-48XP each have a 1280×1024 pixel screen resolution, four hardware color maps in the image planes, and two hardware color maps in the overlay planes.

The HP VISUALIZE-48's and HP VISUALIZE-48XP's device drivers dither all vectors.

PowerShade

The HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices work with the 3D surface rendering software, PowerShade. Note that PowerShade only works in the image planes using the `hphcrx48z` device driver. Rendering support in the overlay planes is provided by the HP VMX driver. For information on this driver, see the "HP Virtual Memory and X" chapter in this Addendum.

PowerShade capabilities are automatically available in the image planes on these devices. In order to use VMX with PowerShade on any graphics system, you must install the PowerShade software.

For More Information

Information provided on the HP VISUALIZE-48 and HP VISUALIZE-48XP is device-specific. For more detailed information on graphics programming and X windows, please refer to the noted documents:

- See the *Starbase Graphics Techniques* manual for general Starbase programming information.
- Refer to the *Graphics Administration Guide* to read about linking shared or archive libraries, path naming conventions, X windows, completing installation, and setting up graphics devices.

Device Description

The HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices have four hardware color maps in the image planes and two hardware color maps in the overlay planes. They support HP Color Recovery in all depth 8 image visuals, as explained in the section "HP Color Recovery Technology" in this chapter.

The HP VISUALIZE-48 and HP VISUALIZE-48XP frame buffers include 48 image planes and 8 overlay planes. Visuals that are supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP are 8/8 double-buffered, 8-plane single-buffered, 24/24 double-buffered, and 24-plane single-buffered. Visuals that are not supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP are 12/12 double-buffered, 12-plane single-buffered, and 48-plane single-buffered. The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer.

You can render to the image planes in two ways:

8-bit color (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`)

24-bit color (`CMAP_FULL`)

The two rendering modes are selected on a per-window basis. The mode selected is a function of the depth of the window created and double-buffer mode.

In 8-bit mode, each pixel is used as an index into a 256-entry color map. Each entry in the color map provides eight bits per color for each of the red, green, and blue components, providing a color palette of over 16 million colors. Double-buffering is achieved by switching between two banks of 8-bit indexes. You can perform 3:3:2 direct color emulation in this mode.

In 24-bit mode, a pixel is represented by eight bits each for red, green, and blue.

There are four hardware color maps available for use with the image planes. All four color maps are shared by all graphics processes.

In addition to the four hardware color maps in image planes, there are two hardware color maps for the overlay planes. One of the hardware color maps has the default X11 color map permanently installed in it. This is done to avoid technicolor in X11 and HP CDE windows and backgrounds.

The X server works only in combined mode. For information on supported X server modes, read the section "Supported X Server Modes" in the *Graphics Administration Guide*.

Geometry Accelerator

The HP VISUALIZE-48 and HP VISUALIZE-48XP include, by default, a geometry accelerator to provide high performance 3D solids modeling and high performance 3D wireframe with anti-aliasing. The HP VISUALIZE-48 and HP VISUALIZE-48XP geometry accelerators have a dedicated 24-bit hardware Z-buffer. The primary use of the HP VISUALIZE-48 and HP VISUALIZE-48XP geometry accelerators is for 3D solids modeling, including drawing Starbase polygons, polyhedrons, rectangles, triangle strips, quadrilateral meshes, spline surfaces, geometry transform, and lighting and shading of primitives. They have capabilities for both surface rendering and volumetric rendering.

Note that the geometry accelerator is not used for rendering in the overlay planes.

The following lists provide information to help you maximize your application performance. The first list describes operations that are most efficient on the HP VISUALIZE-48 and HP VISUALIZE-48XP.

- Isotropic modeling transformations
- Lighting, with no more than 8 lights of any type
- View clipping
- Perspective and orthographic (parallel) transformations
- Depth cueing
- 3- and 4-sided filled primitives, with or without RGB, alpha, and normal data per vertex
- Triangle strips, with or without RGB, alpha, and normal data per vertex
- 2D and 3D polylines

The following features use the geometry accelerator, but yield somewhat lower performance than the base features listed above.

- Non-convex polygons with more than 4 vertices
- Polyhedrons with move/draw flags
- Facet normal lighting
- Facet color

The following features bypass the hardware geometry accelerator and use the PowerShade software renderer instead:

- Self-intersecting polygons
- Model clipping/capping
- Deformation
- Wide lines
- Backface distinguishing (but not back-face culling)
- Starbase `INT_OUTLINE` interior style
- Circles, ellipses, arcs
- `CMAP_NORMAL` or `CMAP_MONOTONIC` modes
- Picking
- `move3d()/draw3d()`
- Polymarkers
- Rectangles
- Text

Note that the geometry accelerator directly handles polygons with 3 or 4 vertices only; more complicated polygons are decomposed into triangles. Convex polygons will be decomposed most easily. Non-convex polygons or fill area sets with only one set will be decomposed less easily. Polyhedrons with move/draw flags will be decomposed, but with a significant penalty in execution time. Self-intersecting polygons can not be decomposed for the geometry accelerator. Instead, they are lighted, shaded, and transformed by PowerShade, with only the final rendering steps performed by the HP VISUALIZE-48 and HP VISUALIZE-48XP scan conversion hardware. Since polygons are decomposed into triangles before transformations occur, visual results may differ slightly from previous devices. Non planar polygons or polygons with greatly differing colors or normals at the vertices will differ more than planar polygons or polygons with more homogeneous vertex data.

Also, note that compound primitives (triangle strips, quadrilateral meshes, and polyhedrons) will perform better than the equivalent multiple discrete polygon calls, since the shared library call overhead is less.

For more information about specific primitives and their relative speeds, read the *Graphics Administration Guide*.

Texture Mapping Accelerator

An optional accelerator for texture mapped primitives may be purchased for use with the HP VISUALIZE-48 and HP VISUALIZE-48XP hardware. Use the `graphinfo` program to determine whether your system has this optional accelerator. The line:

```
texture accelerator:        yes
```

will be present if and only if the texture accelerator hardware is installed.

This hardware accelerates the following texture mapping features:

- Single texture map per primitive
- Full MIP mapping with all MIP interpolation filters
- All post-lighting texturing and pre-light replace and modulate texturing

The accelerator has memory built into it to hold up to 16 megabytes of texture data (with 8-bits red, green, blue, and alpha data per texel). This is enough memory for three 1024×1024 fully MIP-mapped textures, or a single 2048×2048 point sampled texture map. However, through a caching scheme for the hardware texture memory, textures as large as 32768×32768 may be accelerated. Note that up to 4096 textures of size 64×64 or smaller, or 256 textures of size 256×256 or larger can be supported at one time.

To support the HP VISUALIZE-48 and HP VISUALIZE-48XP texture cache, a texture interrupt management daemon runs continuously. This daemon, named `timd`, is responsible for ensuring that the appropriate sections of texture maps reside in the hardware texture memory. Just like other system processes, under no circumstances should you attempt to kill `timd`, as this may cause the hardware to enter a "hung" state from which it is difficult to recover.

Overlay Plane Rendering

Either the `hpvmx` or `sox11` device driver is used for Starbase rendering to the overlay planes. For more information on these device drivers, see the chapters "HP Virtual Memory and X" in this Addendum and "The Starbase-on-X11 Device Driver" in the *HP-UX Starbase Device Drivers Manual*.

If an overlay plane window is `gopen`ed with a driver name of `NULL`, the `hpvmx` driver will be used. See the table, "Driver Selection at gopen" in the chapter "HP Virtual Memory and X" in this Addendum for details.

8/8 VM double-buffering is supported in the overlay planes using the `hpvmx` driver.

HP Color Recovery Technology

The HP VISUALIZE-48 and HP VISUALIZE-48XP use HP Color Recovery for shaded fill areas in depth 8 image-plane visuals (for example, polygons and spline surfaces). Color Recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. HP Color Recovery is available in all depth 8 visuals on the HP VISUALIZE-48 and HP VISUALIZE-48XP.

There are two components to HP Color Recovery. A different dither cell size (16×2) is used when rendering shaded polygons, and a digital filter is used when displaying the contents of the frame buffer to the screen.

The HP VISUALIZE-48 and HP VISUALIZE-48XP provide HP Color Recovery whenever you are in `CMAP_FULL` mode and you have used the `INIT` flag in the `gopen`, `shade_mode`, or the `double_buffer` function to initialize color maps. Keep in mind that the default color map mode is `CMAP_NORMAL` for PseudoColor visuals. Therefore, the HP Color Recovery color map will not be downloaded until you call `shade_mode` to set the mode to `CMAP_FULL` and use `INIT`.

HP Color Recovery is available when using either PseudoColor or TrueColor visuals. The HP Color Recovery color map is a read-only color map. Any attempts to change it will be ignored and no error will be reported.

In `CMAP_FULL` shade mode, disabling HP Color Recovery results in normal dithering of shaded fill areas. HP Color Recovery is not available with any other shade mode.

HP Color Recovery is enabled in conjunction with a particular X color map that is associated with your window. If that X color map is not currently installed in hardware by your window manager, you will not see the effect of the HP Color Recovery filter.

Note that vectors are always dithered, even in an HP Color Recovery window.

Under some conditions HP Color Recovery can produce undesirable artifacts in the image. This also happens with 4×4 dithering, but the artifacts are different. However, images rendered with HP Color Recovery are seldom worse than what dithering produces, and in most cases, HP Color Recovery produces significantly better pictures than dithering. Note that 4×4 dithering, like HP Color Recovery, is available in the `CMAP_FULL` color map mode, but *not* in the `CMAP_NORMAL` color map mode.

HP Color Recovery is available by default. If you wish to disable HP Color Recovery, you can do it in one of three ways:

- Add the screen option `DisableColorRecovery` to your `X*screens` file. Setting this screen option prior to starting up the X server disables HP Color Recovery for *all* applications and any attempts to enable HP Color Recovery will be ignored. Remember, if you set this screen option prior to starting up the X server, you cannot re-enable HP Color Recovery from the command line or from within an application. To set this screen option, add the following lines to your ⟨*x11-admin*⟩[1]`/X0screens` file:

  ```
  ScreenOptions
     DisableColorRecovery
  ```

  and restart HP CDE or X11.
- Export the environment variable `HP_DISABLE_COLOR_RECOVERY` before running your application. Setting this environment variable to any value disables HP Color Recovery for subsequently executed applications. To set this environment variable in your current X11 window, execute this command on the command line before running your application (assuming you are using the Korn shell):

  ```
  export HP_DISABLE_COLOR_RECOVERY=TRUE
  ```
- Disable HP Color Recovery programmatically by using the Starbase `gescape` `COLOR_RECOVERY_CONTROL`. For details on this `gescape`, read the subsequent section "Gescapes."

[1] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Gescapes

The COLOR_RECOVERY_CONTROL gescape can be used to disable HP Color Recovery. Passing it a 0 value in arg1 will disable HP Color Recovery, a 1 value will enable it (HP Color Recovery is enabled by default). The arg2 parameter is ignored. The effect of this gescape will not take place until the next time you call shade_mode or double_buffer with the INIT flag. For example:

```
gescape_arg  arg1;
/* Disable HP Color Recovery */
arg1.i[0] = 0;
gescape(fildes, COLOR_RECOVERY_CONTROL, &arg1, NULL);
shade_mode(fildes, CMAP_FULL|INIT,0);
```

The Frame Buffer

Physical Address Space

The physical frame buffer is addressed as 2048×1024 bytes. The last 768 bytes of each line of the address space (those to the right of the screen) are not displayed and no memory exists in those areas.

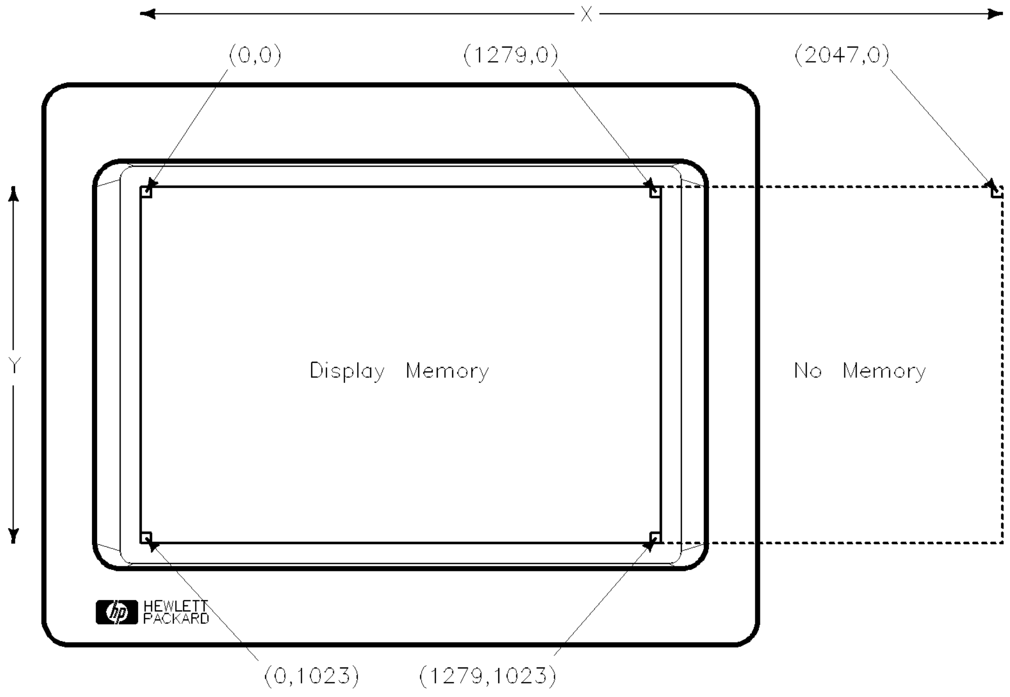

Figure 3-1. Physical Address Space

To Access the Frame Buffer Directly

When using the `R_GET_FRAME_BUFFER` gescape for direct user access to the frame buffer, correct access can only be assured by using the `R_LOCK_DEVICE` and `R_UNLOCK_DEVICE` gescapes.

1. Use `R_LOCK_DEVICE` just prior to direct frame buffer access.
2. Use `R_UNLOCK_DEVICE` directly after the frame buffer access and before any other Starbase commands.

Caution Do not read from or write to the offscreen addresses. Such operations will cause errors.

Frame Buffer Address Mapping

The frame buffer is organized as a single one-dimensional array of pixel values. The first byte (byte 0) of the frame buffer represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1279 is the last (right-most) displayable pixel on the top line. The next 768 bytes are not displayable. Byte 2048 is the first (left-most) pixel on the second line from the top. The last (lower right corner) pixel on the screen is byte number 2,096,383 ($1023\times2048+1279$).

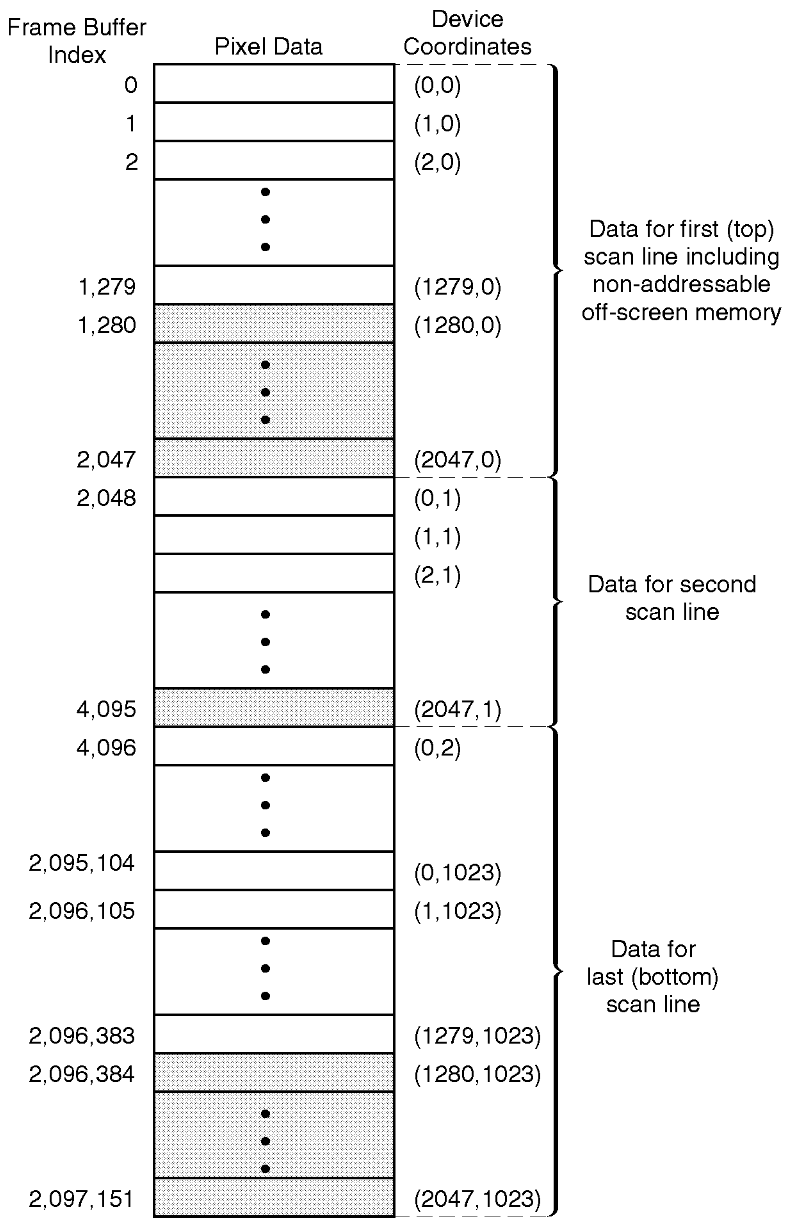

Figure 3-2. Frame Buffer Mapping in Memory

The HP VISUALIZE-48 and HP VISUALIZE-48XP frame buffers have six banks of 8 planes (two for each color). Only one bank can be accessed at a time. Use the `bank_switch` call to select a bank to read or write data directly from the frame buffer. For `block_read` and `block_write` operations to the image planes, the data is in all eight bits of each byte.

The default for reading the Z-buffer is always 24 bits per pixel in a 32-bit word. The `raw` parameter to `block_read` and `block_write` must be set to true in order to read from or write to the Z-buffer. Using `wbank=6` in the `bank_switch` command on the HP VISUALIZE-48 and HP VISUALIZE-48XP selects the Z-buffer for reads or writes.

The Z-buffers for the HP VISUALIZE-48 and HP VISUALIZE-48XP are 24-bits deep, but only 23 bits are available for depth information. When applications read the Z-buffer data, the depth information is returned left-justified in the lower 24 bits of a 32-bit word (that is, the 23-bit Z-buffer data is shifted left one bit from the least-significant bit), as shown in the following figure. This differs from the behaviour of the CRX-48Z.

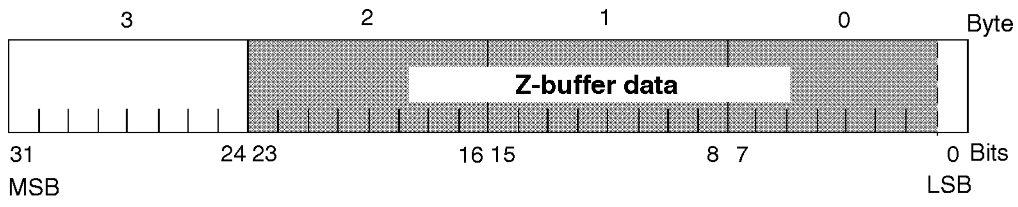

Figure 3-3. Hardware Z-Buffer Data Alignment

Unlike the frame buffer, the Z-buffer data is contiguous. The HP VISUALIZE-48's and HP VISUALIZE-48XP's Z-buffers are always 1280×1024 where word 1280 is the leftmost word of the second scanline. For the HP VISUALIZE-48 and HP VISUALIZE-48XP, the Z-buffer is the size of the window. For example, if the window is 400×400, word 400 is the leftmost Z-buffer value for the second scan line.

Frame Buffer Configurations

The following table shows which color map modes are supported for different frame buffer configurations.

Table 3-2. Supported Frame Buffer Configurations

Number of Planes	HP VISUALIZE-48 **and** HP VISUALIZE-48XP
8/8	`CMAP_NORMAL`, `CMAP_FULL`, `CMAP_MONOTONIC`
24/24	`CMAP_FULL`

Since Starbase supports double-buffering per window, it is better to request double-buffering with a depth of 24 when in `CMAP_FULL` mode on an HP VISUALIZE-48 and HP VISUALIZE-48XP. Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported in depth 8 windows for compatibility with previous devices, however, it is not recommended. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes. Flashing may occur, however, as this kind of double-buffering is not coordinated with the X server.

Using Starbase in X Windows

This section contains device specific information needed to run Starbase programs in X11 windows. If you need a general, device-independent explanation of using Starbase in X11 windows, refer to the "Using Starbase with the X Window System" chapter of *Starbase Graphics Techniques*.

To reduce the complexity of having multiple X server modes, the `hphcrx48z` drivers for X and Starbase only support one X server mode. Several other key features have been designed to improve the overall usability of the devices in the X11 windows environment, and to reduce interaction issues between the X11 user interface and graphics library APIs (such as Starbase), that provide *direct hardware access* (DHA).

Per-Window Double-Buffering

The HP VISUALIZE-48 and HP VISUALIZE-48XP support double-buffering in the images planes on a per-window basis. The HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices support 8/8 and 24/24 planes double-buffered for each of the Starbase color map modes (`CMAP_NORMAL`, `CMAP_FULL`, `CMAP_MONOTONIC`) in the image and overlay planes. Any X11 library drawing routines will render to the currently visible buffer of a window that has double-buffering enabled.

Note that Starbase uses the **hpvmx** device driver to perform double-buffering in software in the overlay planes. This double-buffering method is slower than the hardware double-buffering used in the image planes. Any X11 library drawing routines will render to the currently visible buffer of a window that has double-buffering enabled.

Available Color Map Entries

The HP VISUALIZE-48 and HP VISUALIZE-48XP have two hardware color maps in the overlay planes and four hardware color maps in the image planes.

If you query the X server for the number of entries in the default overlay visual's color map while you are using the default X server mode of the HP VISUALIZE-48 and HP VISUALIZE-48XP, the server will reply that there are 256 entries available. Although all 256 entries are available for use by an application, the last entry (index 255) is not writable because it is allocated by the X server.

Starbase Color Maps and X11 Read/Write Restrictions

The X color model defines read/write restrictions both on color maps and on individual entries in color maps. As of HP-UX 9.05, Starbase no longer overwrites read-only color maps or color map entries as defined in the X color model. Attempts to write to color map entries in read-only color maps (that is, for TrueColor, StaticColor, or StaticGray visuals) are silently ignored.

Accessing HP Color Recovery with X Windows

The HP VISUALIZE-48 and HP VISUALIZE-48XP support HP Color Recovery for shaded areas. When a depth 8 window is used, HP Color Recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. Color Recovery is available on all depth 8 windows on the HP VISUALIZE-48 and HP VISUALIZE-48XP. For more information about HP Color Recovery, read the section "HP Color Recovery" found in this chapter.

The Starbase, HP PEX, and HP-PHIGS graphics libraries provide programmers who use these APIs with transparent access to the HP Color Recovery capability of the HP VISUALIZE-48 and HP VISUALIZE-48XP. If you are producing graphics using Xlib calls, then your application must perform some of the necessary processing. At server start-up, there is one property that is defined and placed on the root window if the `HP_DISABLE_COLOR_RECOVERY` environment variable has *not* been exported. This property is:

```
_HP_RGB_SMOOTH_MAP_LIST
```

The above property is of type `RGB_COLOR_MAP` and carries pointers to structures of type `XStandardColormap`. It may be interrogated with calls to `XGetRGBColormaps`. The property `_HP_RGB_SMOOTH_MAP_LIST` is a list of color maps that are associated with window visual IDs that support HP Color Recovery. When the `XGetRGBColormaps` routine searches throughout this list for a color map with a visual ID that matches your window's visual ID and it finds one, your application knows that your visual supports HP Color Recovery, and uses that color map for any HP Color Recovery window.

HP Color Recovery uses all 256 entries of one of the available color maps. The color visual used by HP Color Recovery emulates the 24-bit TrueColor visual. Thus, the colors red, green, and blue are typically declared as integers in the range from 0 to 255. Note that each window that uses HP Color Recovery will use the same color map.

For HP Color Recovery to produce the best results, the emulated 24-bit TrueColor data is dithered as explained below.

A pixel to be dithered is sent to the routine provided in this example. Note that the values of the variables `RedValue`, `GreenValue` and `BlueValue` are generated by an application. In this example, the color values are assumed to be in the range [0..255].

The given routine receives the color values and the X and Y window address (`Xp` and `Yp`) of the pixel. The X and Y address is used to access the dither tables. The values from the dither tables are added to the color values. After the dither addition, the resultant color values are quantized to 3 bits of red and green and 2 bits of blue. The quantized results are packed into an 8-bit `unsigned char` and then stored in the frame buffer. As the contents of the frame buffer are scanned to the CRT, a special section in the HP VISUALIZE-48 and HP VISUALIZE-48XP hardware then converts the 8-bit data stored in the frame buffer into a 24-bit TrueColor image for display.

Here is a routine that can be used to dither the 24-bit TrueColor data.

```
unsigned char dither_pixel_for_CR(RedValue,GreenValue,BlueValue,Xp,Yp)
int RedValue,GreenValueBlueValue,Xp,Yp;
{
static short dither_red[2][16] = {
  {-16,  4, -1, 11,-14,  6, -3,  9,-15,  5, -2, 10,-13,  7, -4,  8},
  { 15, -5,  0,-12, 13, -7,  2,-10, 14, -6,  1,-11, 12, -8,  3, -9} };

static short dither_green[2][16] = {
  { 11,-15,  7, -3,  8,-14,  4, -2, 10,-16,  6, -4,  9,-13,  5, -1},
  {-12, 14, -8,  2, -9, 13, -5,  1,-11, 15, -7,  3,-10, 12, -6,  0} };

static short dither_blue[2][16] = {
  { -3,  9,-13,  7, -1, 11,-15,  5, -4,  8,-14,  6, -2, 10,-16,  4},
  {  2,-10, 12, -8,  0,-12, 14, -6,  3, -9, 13, -7,  1,-11, 15, -5} };

int red, green, blue;
int x_dither_table, y_dither_table;
unsigned char pixel;

x_dither_table = Xp % 16;   /* X Pixel Address MOD 16 */
y_dither_table = Yp % 2;    /* Y Pixel Address MOD 2 */

red = RedValue;
green = GreenValue;
blue = BlueValue;

if (red >= 48) /* 48 is a constant required by this routine */
   red=red-16;
else
   red=red/2+8;
red += dither_red[y_dither_table][x_dither_table];
if (red > 0xff) red = 0xff;
if (red < 0x00) red = 0x00;

if (green >= 48) /* 48 is a constant required by this routine */
```

```
   green=green-16;
else
   green=green/2+8;
green += dither_green[y_dither_table][x_dither_table];
if (green > 0xff) green = 0xff;
if (green < 0x00) green = 0x00;

if (blue >= 112) /* 112 is a constant required by this routine */
   blue=blue-32;
else
   blue=blue/2+24;
blue += (dither_blue[y_dither_table][x_dither_table]<<1);
if (blue > 0xff) blue = 0xff;
if (blue < 0x00) blue = 0x00;

pixel = ((red & 0xE0) | ((green & 0xE0) >> 3) | ((blue & 0xC0) >> 6));

return(pixel);
}
```

Backing Store

Backing store is only supported when rendering to overlay planes with the `hpvmx` driver. For image plane windows, you need to detect window exposure events and repaint the window when a previously obscured portion of a window is made visible.

The HP VISUALIZE-48 and HP VISUALIZE-48XP support backing store (also known as "retained raster") if acceleration is disabled. The backing store feature allows a window being rendered to by a direct hardware access (DHA) client to be "backed-up" to a virtual frame buffer whenever any portion of the window is obscured by another window. In this case, the application is not required to catch "expose events" from X11 and redraw the picture when occlusion occurs. In fact, no "expose events" will be generated if backing store is enabled.

Thus, when a window is placed on top of another window containing a complete image, the window system will save the contents of the latter window before displaying the obscuring window. Then, when the obscuring window is removed, the earlier contents of the occluded area plus any new rendering that has occurred in the occluded area during the cover-up will be restored. Since rendering to the virtual frame buffer is not as fast as rendering to the actual frame buffer in the occluded area, the performance will suffer, but only while the window is occluded.

Backing Store Exceptions

In general, those Starbase operations that draw to the display are also supported when drawing to backing store. The exceptions to this are:

- Backing store with 24-plane visuals is not supported.
- Backing store for the HP VISUALIZE-48 and HP VISUALIZE-48XP accelerators cannot be enabled.
- Backing store will not work with certain `gescape` operations that access device-dependent features.
- Backing store contents may be incorrect if you mix Xlib rendering with Starbase rendering to an 8/8 double-buffered window.

If these limitations on backing store support prove troublesome in your application, do not use backing store. Instead, detect window exposure events and repaint the window when a previously obscured portion of a window is made visible.

X11 Cursor

The X11 cursor (often called the sprite) is maintained by the display hardware and never interferes with the frame buffer contents in either the image or overlay planes.

Supported Visuals

The following table of Supported Visuals contains information for programmers using either Xlib graphics or Starbase. The table lists the image plane depths of windows and color map access modes that are supported for a given graphics device. It also indicates whether or not backing store (also known as "retained raster") is available for a given visual, and lists the double-buffer configurations supported by Starbase for this device driver.

Table 3-3. Supported Visuals

Device	Depth	Visual Class	Backing Store		Starbase Double-Buffer[1]
			Xlib	Starbase	
HP VISUALIZE-48, HP VISUALIZE-48XP	8	PseudoColor	Yes[2]	No[3]	8/8
		TrueColor	Yes[2]	No[3]	8/8
	24	DirectColor	Yes	No	24/24
		TrueColor	Yes	No	24/24

1 Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported for compatibility with previous devices, however, it is not recommended. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes in a depth 8 visual. Flashing may occur, however, as this kind of double-buffering cannot be done on a per window basis. Note that double-buffering with less than 8-planes is *only* supported in `CMAP_NORMAL`.

2 Xlib primitives are supported by backing store. Whenever backing store is not maintained, normal expose events are generated.

3 Backing store is only supported when rendering with the `hpvmx` driver.

Moving the Default Visual to the Image Planes

Note By default the overlay planes have the default X11 color map permanently locked into one hardware color map, and any other hardware color maps used by the overlay planes are available for applications to use. Moving the default visual into the image planes will limit the number of hardware color maps available to you. In this mode, the HP VISUALIZE-48 and HP VISUALIZE-48XP provide a single hardware color map in the overlay planes.

Since HP-UX 9.05, X Windows have provided a method for changing the default visual from a depth 8 overlay PseudoColor visual to a depth 8 image PseudoColor visual. This is done by moving the default visual to the depth 8 image PseudoColor visual. To do this, use SAM. Or, manually edit the file[2]:

```
⟨x11-admin⟩/X*screens
```

and add the following lines:

```
Screen ⟨dev⟩/crt
   DefaultVisual
      Class PseudoColor
      Depth 8
      Layer Image
```

The `*` in the `X*screens` file name specifies the *display_number*. To determine the display number, execute this shell command:

```
echo $DISPLAY
```

Your results will have the following syntax:

```
⟨host_name⟩:⟨display_number⟩.⟨screen_number⟩
```

[2] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Here is an example of what your display name might look like after executing the `echo $DISPLAY` shell command:

```
mysystem:0.0
```

where *host_name* is `mysystem`, *display_number* is 0, and *screen_number* is 0. In the above example, you would edit the file:

⟨*x11-admin*⟩`/X0screens`

Note that the syntax of this specification has changed. For more information, see the file:

⟨*x11*⟩`/Xserver/info/screens/hp`

Device Support for the TrueColor Visual

TrueColor Visual Description

A TrueColor visual can be thought of as having a read-only color map where, for any given pixel value, about one third of the bits are used to describe each of the red, green, and blue colors, respectively. For an 8-plane TrueColor visual, 3 bits describe the red component, 3 bits describe the green component, and 2 bits describe the blue component. A 24-plane TrueColor visual uses 8 bits each to describe the red, green, and blue components. This is illustrated as follows:

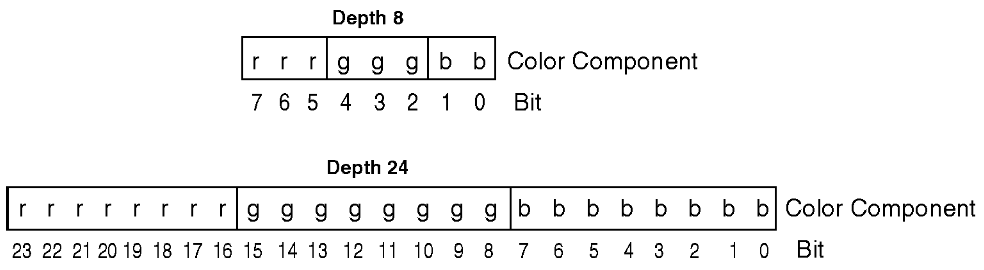
Depth 8
r r r g g g b b Color Component
7 6 5 4 3 2 1 0 Bit
Depth 24
r r r r r r r r g g g g g g g g b b b b b b b b Color Component
23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Bit

Figure 3-4. Pixel Representation for the 8- and 24-Plane TrueColor Visuals

The following example refers to an depth 8 TrueColor visual; however, the example can be expanded to apply to depth 24 plane TrueColor visuals.

Example

Since the red and green components consist of 3 bits each, there are 8 different shades of red and 8 different shades of green available. There are 4 different shades of blue represented by 2 bits. As the component value increases, the intensity of that color increases. For example, a red component of 000 represents no red and a red component of 111 represents full red. Therefore, pixel value 0 is 000 red, 000 green, and 00 blue, which results in black. Pixel value 255 is 111 red, 111 green, and 11 blue, which results in white. These and other examples are shown in Table 3-4.

Table 3-4. Examples of Pixel Color Values

Pixel Value	Binary	Red	Green	Blue
0	000 000 00	shade 0	shade 0	shade 0
53	001 101 01	shade 1	shade 5	shade 1
139	100 010 11	shade 4	shade 2	shade 3
218	110 110 10	shade 6	shade 6	shade 2
255	111 111 11	shade 7	shade 7	shade 3

Note that the red, green and blue intensities for the color shades increase uniformly between 0 and 255.

Device Specific Visuals Information

Note that the TrueColor Visual always uses a `shade_mode` of `CMAP_FULL`. When the TrueColor visual window is `gopen`ed, your application will automatically be in `CMAP_FULL` mode and the `shade_mode` call will ignore any attempts to go into another mode.

With the addition of the TrueColor visual at the 9.03 release of HP-UX, you need to consider the following information:

- If you originally created your windows using a command line similar to the following:

  ```
  xwcreate -g 600x500 -depth 24 window
  ```

  you will have to change how you create your windows by using command lines similar to this:

  ```
  xwcreate -g 600x500 -depth 24 -visual TrueColor window
  xwcreate -g 600x500 -depth 24 -visual DirectColor window
  ```

  Note the addition of the command line option `-visual` for declaring TrueColor and DirectColor visuals. The HP VISUALIZE-48 and HP VISUALIZE-48XP devices default to PseudoColor if the visual is not specified.

- The TrueColor color map is read-only, so it cannot be modified. Note that any attempt to modify the TrueColor color map will not produce an error message.

  One class of applications that could be effected by this are those that perform their own gamma correction.

- If your application searches for a visual by traversing the visual list returned by the X server, you will find that the order of visuals in this list has changed because of the addition of the TrueColor visuals. Therefore, your application code should always explicitly search for a particular visual rather than assuming that it occurs in a fixed position within the list of visuals returned by X11.

The Overlay Plane Color Map Management Scheme

Many applications use the default X11 color map. A technicolor effect (color flashing) in the windows using the default color map occurs when a non-default color map is downloaded into the hardware color map that had previously contained the default X11 color map.

Because so many applications use the default X11 color map, and because the HP VISUALIZE-48 and HP VISUALIZE-48XP have two hardware color maps in the overlay planes, the behavior on these devices is to dedicate (that is, lock) one overlay hardware color map to always hold the default X11 color map. This means that the assigned default overlay hardware color map cannot have another color map downloaded to it. The other overlay hardware color map is available to applications that use color maps other than the default.

Overlay Plane Transparency and the X Windows System

The default X11 mode on the HP VISUALIZE-48 and HP VISUALIZE-48XP do not provide an overlay visual with a transparent property. If you need an overlay color map that supports transparency, create the color map using the visual that has transparency in its `SERVER_OVERLAY_VISUALS` property (see the next section).

An overlay visual's transparency feature enables you to render opaque objects (for example, menus and text) to a transparent overlay window and at the same time view rendered objects in an image window. For example, you may want to show a map of a country without all of its internal borders, and then add the internal borders as you need them. This can be done by creating two X windows with the same geometry: one in the overlay planes and one in the images planes. The country's terrain and boundaries would be drawn in the image planes window and the internal borders in a transparent overlay window.

The following section describes the default frame buffer configuration for the HP VISUALIZE-48 and HP VISUALIZE-48XP.

- The default visual configuration is:
  - overlay planes
  - depth 8
  - PseudoColor
  - opaque (no transparency)

  with 256 color map entries. Note that when using the default X server mode of the HP VISUALIZE-48 and HP VISUALIZE-48XP, if you query the X server for the number of entries in the default color map, the server will reply that there are 256 entries available. Although these entries are available, the X server reserves the last entry (index 255). So, that entry is not writable and should not be used.
- The default X11 color map is locked into one of the hardware color maps in the overlay planes. For a description of how to move the default visual to images planes, read the subsequent section "Moving the Default Visual to the Images Planes" in this chapter.

In the default overlay visual, the number of overlay color map entries is 255 because the last entry is the transparent color map value. If your application requires that you have 256 entries in your color map, you need to set the environment variable `CountTransparentInOverlayVisual`. To do this, use SAM. Or you can manually edit the ⟨*x11-admin*⟩[3]`/X*screens` file to add the following option:

```
ScreenOptions
    CountTransparentInOverlayVisual
```

before starting the X11 server. Any attempts to modify entry 255 will silently ignored, and will have no effect on the color map.

[3] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

The Default Frame Buffer Configuration for the HP VISUALIZE-48 and HP VISUALIZE-48XP

Table 3-5 shows the default HP VISUALIZE-48 and HP VISUALIZE-48XP frame buffer configurations. These configurations are not changed by using overlay transparency.

Table 3-5.
The Default Frame Buffer Configuration for the HP VISUALIZE-48 and HP VISUALIZE-48XP

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	2	no	PseudoColor[1]
overlay	8	single	2	yes	PseudoColor
image	8	single or double	4	N/A	PseudoColor[2] TrueColor
image	24	single or double	4	N/A	DirectColor TrueColor

1 This is the default overlay visual.

2 This is the first visual returned by `xdpyinfo`.

If you need an overlay color map that supports transparency, create the color map using the visual that has transparency in its `SERVER_OVERLAY_VISUALS` property. To look at the contents of this property, you would use code similar to the following:

```
/* First, get the list of visuals for this screen. */
    .
    .
*pVisuals = XGetVisualInfo(display, mask, &getVisInfo, numVisuals);
    .
    .
/* Now, get the overlay visual information for this screen.  To obtain
 * this information, get the SERVER_OVERLAY_VISUALS property. */

overlayVisualsAtom = XInternAtom(display, "SERVER_OVERLAY_VISUALS", True);
if (overlayVisualsAtom != None)
{
        /* Since the Atom exists, we can request the property's contents. */
        bytesAfter = 0;
        numLongs = sizeof(OverlayVisualPropertyRec) / 4;
        do
        {
            numLongs += bytesAfter * 4;
            XGetWindowProperty(display, RootWindow(display, screen),
                               overlayVisualsAtom, 0, numLongs, False,
                               overlayVisualsAtom, &actualType, &actualFormat,
                               &numLongs, &bytesAfter, pOverlayVisuals);
        } while (bytesAfter > 0);
    }
        .
        .
    /* Process the pOverlayVisuals array. */
    while (--nVisuals >= 0) {
        nOVisuals = *numOverlayVisuals;
        pOVis = *pOverlayVisuals;
        imageVisual = True;
        while (--nOVisuals >= 0) {
            pOOldVis = (OverlayVisualPropertyRec *) pOVis;
            if (pVis->visualid == pOOldVis->visualID)
            {
                imageVisual = False;
                pOVis->pOverlayVisualInfo = pVis;
                /* Found the transparent visual */
                if (pOVis->transparentType == TransparentPixel);
            }
            pOVis++;
        }
    }
```

This program segment is not complete; however, its main purpose is to give you an idea of how a visual is checked for overlay transparency. The source for the above code can be found in the file[4]:

⟨*sb-utils*⟩`/wsutils.c`

[4] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

To Open and Initialize the Device for Output

Syntax Examples

C programs[5]**:**

```
fildes = gopen("⟨screen⟩/window", OUTDEV, NULL, INIT);
```

FORTRAN77 programs:

```
  fildes = gopen('⟨screen⟩/window'//char(0), OUTDEV,
+                char(0), INIT)
```

Pascal programs:

```
fildes := gopen('⟨screen⟩/window', OUTDEV, '', INIT);
```

Parameters for gopen

The `gopen` procedure has four parameters:*path*, *kind*, *driver*, and *mode*.

- *path* — This is the name of the device file created by `xwcreate`(1) or created with `XCreateWindow`(3X) and returned from `make_X11_gopen_string`(3G).
- *kind* — This parameter should be `OUTDEV` if the window will be used for output, `INDEV` if the window will be be used for Starbase input, or `OUTINDEV` if the window will be used for both output and Starbase input.
- *driver* — The character representation of the driver type. If this parameter is set to `NULL`, then `gopen` will inquire the device and use the appropriate driver. Where there are both accelerated and unaccelerated versions of the driver, the default is to load the accelerated version.

  For example:

`NULL`	*for C.*
`char(0)`	*for FORTRAN 77.*
`''`	*for Pascal.*

[5] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Or, a character string may be used to specify a driver. For example:

`"hphcrx48z"`	*for C.*
`'hphcrx48z'//char(0)`	*for FORTRAN 77.*
`'hphcrx48z'`	*for Pascal.*

- *mode* — The mode control word consists of several flag bits *OR*ed together. Listed below are flag bits that have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure. See the *Starbase Graphics Techniques* manual for a description of `gopen` actions when accessing an X11 Window.
  - `0` (zero) — Open the device, but do nothing else. The software color table is initialized from the current state of the hardware color map.
  - `INIT` — Open and initialize the device as follows:
    1. The frame buffer is cleared to zeros.
    2. The color map is reset to its default values.
    3. The display is enabled for reading and writing.
    4. The Z-buffer is cleared.
  - `RESET_DEVICE` — Same as INIT.
  - `SPOOLED` — Not supported; raster devices cannot be spooled.
  - `MODEL_XFORM` — Opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
  - `INT_XFORM` — Perform only integer and common operations. All floating point operations will cause an error.
  - `INT_XFORM_32` — Perform only integer and common operations, with extended precision. All floating point operations will cause an error.

    This mode is provided for compatibility of integer precision with previous devices. `INT_XFORM` might use a faster transformation pipeline with slightly less precision. It is recommended to use `INT_XFORM` unless maximum precision is required. If maximum precision is required, even at the expense of performance, use `INT_XFORM_32`.
  - `FLOAT_XFORM` — Perform only floating point and common operations. All integer operations will cause an error.

- `UNACCELERATED` — Tells the `hphcrx48z` driver to not use the HP VISUALIZE-48 and HP VISUALIZE-48XP accelerators. This flag only applies when `NULL` is specified for the driver name.
- `ACCELERATED` — Tells the `hphcrx48z` driver to use the HP VISUALIZE-48 and HP VISUALIZE-48XP accelerators (default). This flag only applies when `NULL` is specified for the driver name.

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location $(0, 0)$ is the upper-left corner of the window with X-axis values increasing to the right and Y-axis values increasing down.

Use this form of pixel addressing when calling high-level Starbase operations in terms of (`x,y`) device coordinates.

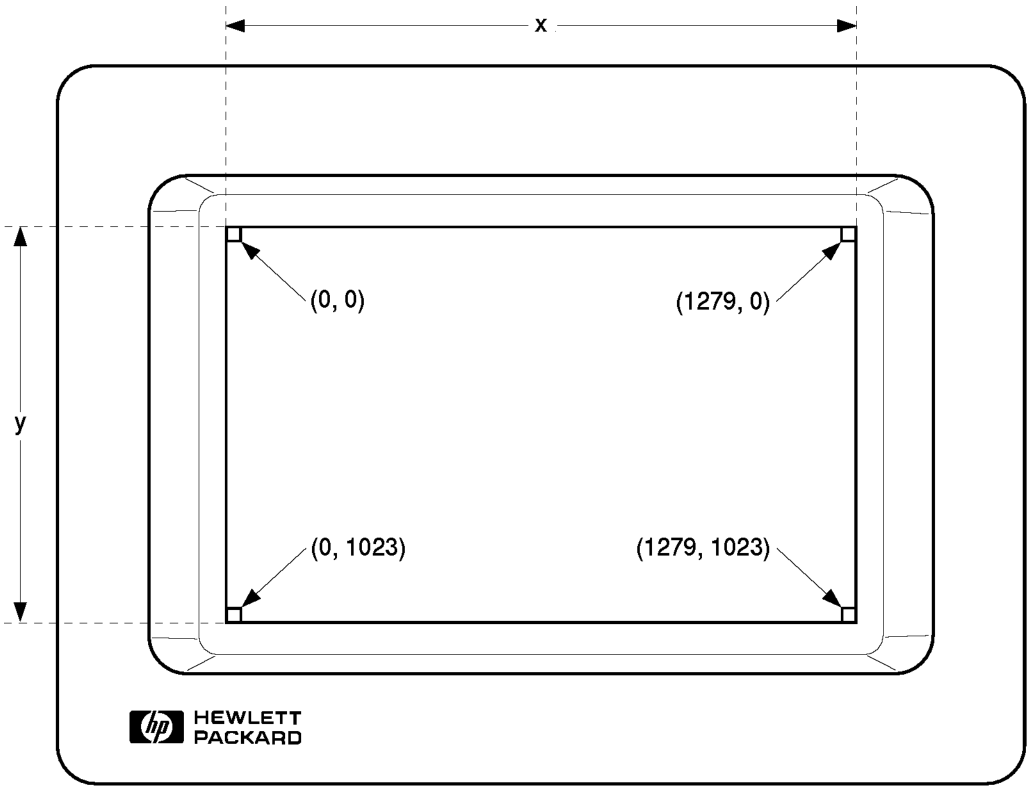

Figure 3-5. Device Coordinates

Starbase Echoes

This section provides information about the echo implementation for Starbase. The HP VISUALIZE-48, HP VISUALIZE-48XP, and all future graphics device drivers will use this implementation of Starbase echoes.

Starbase echoes use Xlib functionality to draw echoes in the same planes as the visual that is active for the window. All previously supported Starbase echo functions are implemented except for those listed below:

- There is no support for the following gescapes:

  ```
  R_DEF_ECHO_TRANS
  R_ECHO_FG_BG_COLORS
  R_OV_ECHO_COLORS
  R_OVERLAY_ECHO
  ```

  In addition, the `R_ECHO_MASK` is only supported for a maximum of two colors within the raster definition.
- More than one Starbase echo per window is not supported.
- Rendering with both Xlib and Starbase in the same window while Starbase echoes are active may produce some random pixel "noise".

Starbase Functionality

This section contains information on Starbase calls that are *not* supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP. This section also contains information on `gescapes` that are supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP.

Calls Not Supported on the HP VISUALIZE-48 and HP VISUALIZE-48XP

The following calls are not supported when using the HP VISUALIZE-48 and HP VISUALIZE-48XP:

```
bf_texture_index
contour_enable
define_contour_table
define_texture
texture_index
texture_viewport
texture_window
```

Conditional Support of Starbase Calls on the HP VISUALIZE-48 and HP VISUALIZE-48XP

The following calls are supported with the listed exceptions:

`alpha_transparency`	The HP VISUALIZE-48 and HP VISUALIZE-48XP support alpha transparency. Alpha only applies to filled areas such as polygons, quadrilateral meshes, triangular strips, and spline surfaces. Vector primitives are not rendered with `alpha_transparency`. The alpha transparency feature is limited to `CMAP_FULL` in a depth 24 visual. Only the floating point version of these primitives will be rendered with alpha transparency; device coordinate primitives do NOT use alpha. The HP VISUALIZE-48 and HP VISUALIZE-48XP do not support alpha transparency with attenuation. (See `alpha_transparency`(3G) in the *Starbase Reference* manual for the list of parameters).

`block_read`,
`block_write`

The *raw* parameter for the `block_read` and `block_write` commands is used by this driver to do plane-major reads and writes. It is enabled by the gescape `R_BIT_MODE`.

The storage destination supplied by the user as the source or destination must be organized as follows.

- The data from each plane is packed with eight pixels per byte.
- Each row must begin on a byte boundary. Thus, the size of the rectangle as specified by the $\langle length\_x \rangle$ and $\langle length\_y \rangle$ parameters must correspond to an integral number of bytes.
- The data for the next plane begins on the following byte boundary.
- Clip to the screen limits.
- The first pixel in the source rectangle is placed in the high-order bit of the first byte in each plane region.
- When clipping, part of each plane region will not be read (`block_read`) or altered (`block_write`).

A bit mask selects the planes to read or write. The initial value of this mask is 1 (one) indicating that only plane 0 is to be accessed. The value of the mask may be changed using the `R_BIT_MASK` or `GR2D_PLANE_MASK` gescapes. `GR2D_PLANE_MASK` is discussed in the appendix of this manual. The planes selected by the mask are expected to reside in consecutive plane locations in the user storage area. This reduces the storage requirements to exactly what is needed but also presents the potential for addressing violations or undesirable results.

For example, if the plane mask is changed to specify more planes between a `block_read` and a following `block_write` from the same location, the `block_write` will attempt to access storage for planes that were not read (and perhaps not allocated). The application program must ensure consistency in these operations.

`fill_dither`	The ability to dither is disabled if the number of colors specified by `fill_dither` is one. However, if the number of colors specified is greater than 1, the default dither cell size of 16 is used. Dithering is only used in depth 8 windows while in either the `CMAP_FULL` or `CMAP_MONOTONIC` color map mode.
`interior_style`	The *style*s `INT_PATTERN` and `INT_HATCH` are not supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices.
`light_source`	Up to 15 directional light sources are available on the HP VISUALIZE-48 and HP VISUALIZE-48XP. These devices' hardware accelerates up to eight directional light sources. Using nine to fifteen directional light sources will cause a noticeable performance degradation.
`line_filter`, `perimeter_filter`	Anti-aliasing is supported on the HP VISUALIZE-48 and HP VISUALIZE-48XP. Anti-aliasing for this device applies only to floating point vectors. Device coordinate primitives do not use anti-aliasing. The anti-aliasing features are also limited to the `CMAP_FULL` color map mode in a depth 24 visual.

The HP VISUALIZE-48 and HP VISUALIZE-48XP have two anti-aliasing modes that may be specified with the `line_filter` and `perimeter_filter` procedures. The index values are assigned as follows:

`0` Anti-aliasing disabled, all vectors have one pixel wide output.

	1 Anti-aliasing enabled, all vectors have two pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according the the formula: $pixel\_color$ = ($new\_pixel \times \alpha$) + ($old\_pixel \times$ (1 - α));
	2 Anti-aliasing enabled, all vectors have two pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according to the formula: $pixel\_color = (new\_pixel \times \alpha) + old\_pixel$
	Note that this implementation of 2 pixel wide anti-aliasing differs from the CRX-48Z (with 3-wide anti-aliasing).
`shade_mode`	The color map mode may be selected. Shading can be turned on only if using PowerShade. Shading is not supported on device coordinate primitives even with PowerShade. Note that the HP VISUALIZE-48 and HP VISUALIZE-48XP automatically use PowerShade.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	If using PowerShade software, `vertex_format` is fully functional. Note that the HP VISUALIZE-48 and HP VISUALIZE-48XP automatically use PowerShade.

`*_with_data`

The following routines are called `with_data` routines because they allow you to send extra vertex data. These `with_data` routines are supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP.

```
partial_polygon_with_data3d
polygon_with_data3d
polyhedron_with_data
polyline_with_data3d
polymarker_with_data3d
polyquad_with_data3d
polytriangle_with_data3d
quadrilateral_mesh_with_data
triangle_strip_with_data
```

Note that the `TEXTURE_MAP` flag applies to the TurboVRX devices via the `texture_*` routines. This extra data per vertex is not used in the `tm_*` routines.

For detailed information on these routines, read the *Starbase Reference* and "Appendix A" of this document. In some cases, you will be able to find the routines under their own name, but in other cases, you will need to use the first part of the routine name to locate these routines (e.g., `polyline_with_data3d` is described on the man page for `polyline(3G)`).

Supported Gescapes

The `hphcrx48z` device driver supports the following gescape operations on the HP VISUALIZE-48 and HP VISUALIZE-48XP configurations. Refer to Appendix A of the *HP-UX Starbase Device Drivers Manual* for details on gescapes.

- `BLOCK_WRITE_SKIPCOUNT`—Specify byte skip count during block write.
- `COLOR_RECOVERY_CONTROL`—Disable HP Color Recovery.
- `CUBIC_POLYPOINT`—Specify points to be rendered in a cubic volume specified in modeling coordinates.
- `DC_PIXEL_WRITE`—Specify points to be rendered along a horizontal scan line.
- `DRAW_POINTS`—Select different modes of rounding for rendered points.
- `GAMMA_CORRECTION`—Enable/disable gamma correction.
- `GCRX_PIXEL_REPLICATE`—Pan and zoom a raster image.
- `IGNORE_RELEASE`—Trigger only when button pressed.
- `ILLUMINATION_ENABLE`—Turn on/off illumination bits.
- `LINEAR_POLYPOINT`—Specify points to be rendered along a line specified in modeling coordinates.
- `LS_OVERFLOW_CONTROL`—Set light source overflow handling.
- `POLYGON_TRANSPARENCY`—Segment control over front/back face screen.
- `READ_COLOR_MAP`—Read Color Map.
- `R_BIT_MASK`—Bit mask.
- `R_BIT_MODE`—Bit mode.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_LINE_TYPE`—User defined line style and repeat length.
- `R_LOCK_DEVICE`—Lock device.
- `R_UNLOCK_DEVICE`—Unlock device.
- `TRIGGER_ON_RELEASE`—Trigger only when button is released.
- `SET_POLYGON_OFFSET`—Enable Z-buffer biasing of fill pixels.
- `STEREO`—Activate stereo output mode.
- `SWITCH_SEMAPHORE`—Semaphore Control.
- `TRANSPARENCY`—Set screen door transparency mask (front face and back face).
- `WIDELINE_CONTROL`—Turn on/off and set attributes of widelines.
- `ZBANK_ACCESS`—Enable/disable Z-buffer block operations.
- `ZWRITE_ENABLE`—Enable/disable replacement of Z value.

Exceptions to Gescape Support

Note Because gescape operations are device-dependent, the exceptions discussed below may be removed in future drivers.

`GAMMA_CORRECTION`	Gamma correction is implemented by modifying the color map. It is available only in 24-bit DirectColor visuals on the HP VISUALIZE-48 and HP VISUALIZE-48XP. For information on the gescape `GAMMA_CORRECTION`, refer to Appendix A in the *HP-UX Starbase Device Drivers Manual*. If a global gamma correction value has been set via the X server or the gamma correction tool, that global gamma correction value will be used and the color map will not be modified. See the *Graphics Administration Guide* for more information about the gamma correction tool.

When the gescape operations listed below are used with a backing store graphics window, they will have the desired effect for the visible portion of the window, but may cause the backing store for obscured parts to be altered in inconsistent ways. The features involved (along with the names of the affected gescape operations) are listed below. For more details on the gescape operations, refer to Appendix A in the *HP-UX Starbase Device Drivers Manual*.

`R_BIT_MASK`	The gescape operation `R_BIT_MASK` defines a plane mask to the driver that is used for bit/pixel access to a single plane in the frame buffer. As with other device drivers, only the plane corresponding to the highest bit set in the mask is transferred.
`R_BIT_MODE`	When calling `block_read` or `block_write` with the *raw* parameter set to `TRUE`, the driver supports bit/pixel frame buffer access to single planes.

Modified Gescapes

The HP VISUALIZE-48 and HP VISUALIZE-48XP support the `Z_CLIP_VOXEL` flag for the `CUBIC_POLYPOINT` and `LINEAR_POLYPOINT` gescapes. This flag enables read-only Z-clipping of the point primitives against the existing contents of the Z-buffer. The Z-buffer is not modified.

For CUBIC_POLYPOINT, the Z_CLIP_VOXEL flag is one of several in the vertex format argument, arg1.i[8]. For LINEAR_POLYPOINT, the Z_CLIP_VOXEL flag is one of several in arg1.i[8]. Consult the gescape chapter for more information about LINEAR_POLYPOINT and CUBIC_POLYPOINT.

Comparison Between the CRX-48Z/HCRX-24Z and the HP VISUALIZE-48/HP VISUALIZE-48XP

HP VISUALIZE-48 and HP VISUALIZE-48XP use the hphcrx48z device driver instead of the hpcrx48z and hphcrx device drivers used by the CRX-48Z and HCRX-24Z graphics devices.

The HP VISUALIZE-48 and HP VISUALIZE-48XP are highly compatible with the CRX-48Z and HCRX-24Z graphics devices. This allows applications written and delivered for the CRX-48Z and HCRX-24Z to use the HP VISUALIZE-48 and HP VISUALIZE-48XP accelerators without requiring different executable code.

The gescapes available for the CRX-48Z and HCRX-24Z are supported by the HP VISUALIZE-48 and HP VISUALIZE-48XP.

Possible behavioral differences between the CRX-48Z and HCRX-24Z and the HP VISUALIZE-48 and HP VISUALIZE-48XP are mostly because of hardware differences. These behavioral differences should not affect the operation of the application and may only be observed when directly comparing the images between the accelerated and unaccelerated driver. Some of these differences are discussed below.

Backing Store

Backing store is an X11 feature that allocates main memory for obscured regions of a window. Graphics operations are written to this memory as well as the screen. When the window is unobscured, the screen is updated from this memory. This feature is not supported by the HP VISUALIZE-48, HP VISUALIZE-48XP, CRX-48Z, and HCRX-24Z, unless rendering to the overlay planes. Therefore, applications in the X environment should capture and act on expose events and redraw the image when one is received. X events are documented in the *Programming with Xlib* manual.

Note Support for both backing store and save under may change in future releases of the Starbase graphics library.

Image Differences

Because of the different mechanisms used to generate the image when using the HP VISUALIZE-48 and HP VISUALIZE-48XP accelerators, there may be minor visual differences between accelerated and unaccelerated images. These minor differences are listed below.

- Different visibility properties - very small primitives may be invisible, that is, where the rendering starts and stops on the same pixel, such as the dot on the letter i. If every pixel is required, see the `DRAW_POINTS` gescape.
- Slight shifts in the image location on the display.
- Minor differences in color interpolation.
- Minor differences in pattern alignment or line type segment alignment.

Note These differences should be minor. They may change in future releases of the Starbase graphics library.

`screenpr` for the HP VISUALIZE-48 and HP VISUALIZE-48XP

The *screenpr(1G)* command has been designed to use the X11 and the HP imaging library, rather than Starbase, to read to the screen that is displayed by an HP VISUALIZE-48 or HP VISUALIZE-48XP. This implementation of *screenpr(1G)* correctly processes image and overlay planes, multiple color maps, and overlay transparency.

Note that command line options are still the same; however, if `screenpr` detects it is running on a HP VISUALIZE-48 or HP VISUALIZE-48XP, it will use the `DISPLAY` environment variable to determine the screen to read, rather than using the device file path given by the `-F` option.

The `-p` option to print a single plane (and consequently the `-f` and `-b` options) is not supported on the HP VISUALIZE-48 and HP VISUALIZE-48XP versions of `screenpr`. These options will be ignored by `screenpr`.

This new version of `screenpr` uses X11 image library calls and executes *pcltrans(1G)* to produce PCL output. Therefore, `screenpr` may produce error messages from X, the HP imaging library, or `pcltrans`.

The HP VISUALIZE-48 and HP VISUALIZE-48XP `screenpr` implementations always expands the data to 24 bits. Therefore, the PCL output of an 8-bit only device will be approximately 3 times larger than might be expected.

The HP Visualize-FX Family of Devices

The `hpvisx` driver supports the HP VISUALIZE-FX family of graphics devices. This family includes the following devices:

- HP VISUALIZE-FX^2
- HP VISUALIZE-FX^4
- HP VISUALIZE-FX^6

Anywhere "HP VISUALIZE-FX" or "the HP VISUALIZE-FX family" is used in this documentation, it refers to all the graphics devices in the above list.

The HP VISUALIZE-FX devices provide hardware support for the following operations:

- Generating vectors
- Wide lines
- Write-enabling planes and selecting individual banks in the frame buffer
- Writing pixels to the frame buffer with a given replacement rule
- Moving a block of pixels within the frame buffer
- Double-buffering per window
- Flat shaded rectangles
- Dithering and HP Color Recovery technology
- Overlay plane transparency (X Windows)
- Gouraud Shading
- Positional and directional lighting calculations
- Anti-aliased lines
- Alpha transparency
- Hidden surface removal with Z-buffer
- Z-buffering of voxels
- Model geometry transformations
- Optional hardware acceleration of texture mapping
- Polygon offset

The `hpvisx` device driver supports the high performance graphics devices described in subsequent sections.

Graphics Device Configuration

Note The HP VISUALIZE-FX graphics devices do not provide raw-mode graphics support. You must display your Starbase applications in an X11 window or windows. For information on using Starbase with X11, read the chapter "Using Starbase with the X Window System" in the *Starbase Graphics Techniques* manual.

The HP VISUALIZE-FX devices support the configurations shown in Table 4-1.

Table 4-1. Supported Depth of Image Plane Windows

Graphics Devices	Number of Image Planes	Number of Overlay Planes	Hardware Accelerator	Resolution
HP VISUALIZE-FX^2	8/8, 12/12, or 24	8[1]	Yes	1280×1024
HP VISUALIZE-FX^4, HP VISUALIZE-FX^6	8/8 or 24/24	8[1]	Yes	1280×1024

1 All Starbase graphics rendering to the overlay planes is done by the VMX or SOX11 device driver.

In order to reduce flickering, this graphics device refreshes the attached CRT displays at a 72 Hz rate.

The HP VISUALIZE-FX devices each have a 1280×1024 pixel color display, four hardware color maps in the image planes, and two hardware color maps in the overlay planes.

Note Starbase supports *only* a 1280×1024 resolution display that refreshes at 72 Hz for the HP VISUALIZE-FX devices. No other configurations are supported by Starbase.

PowerShade

The HP VISUALIZE-FX graphics devices work with the 3D surface rendering software, PowerShade. Note that PowerShade only works in the image planes using the `hpvisx` device driver. Rendering support in the overlay planes is provided by the HP VMX driver. For information on this driver, see the "HP Virtual Memory and X" chapter in this Addendum.

PowerShade capabilities are automatically available in the image planes on these devices. In order to use VMX with PowerShade on any graphics system, the PowerShade software must be installed.

For More Information

Information provided on the HP VISUALIZE-FX family of graphics devices is device-specific. For more detailed information on graphics programming and X windows, please refer to the noted documents:

- See the *Starbase Graphics Techniques* manual for general Starbase programming information.
- Refer to the *Graphics Administration Guide* to read about linking shared or archive libraries, path naming conventions, X windows, completing installation, and setting up graphics devices.

Device Descriptions

The information contained in this section is true for all HP VISUALIZE-FX devices. See the sections "HP VISUALIZE-FX^2 Device Description" and "HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 Device Descriptions" for device-specific information.

Each graphics device in the HP VISUALIZE-FX family has four hardware color maps in the image planes and two hardware color maps in the overlay planes. They support HP Color Recovery in all depth 8 image visuals, as explained in the section "HP Color Recovery Technology" in this chapter.

You can render to the image planes in three ways:

8-bit color (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`) on all HP VISUALIZE-FX devices (HP VISUALIZE-FX[2], HP VISUALIZE-FX[4], and HP VISUALIZE-FX[6])

12-bit color (`CMAP_FULL`) on HP VISUALIZE-FX[2] only

24-bit color (`CMAP_FULL`) on all HP VISUALIZE-FX devices (24-plane single-buffered on HP VISUALIZE-FX[2] and 24/24 double-buffered on HP VISUALIZE-FX[4] and HP VISUALIZE-FX[6])

The rendering mode is selected on a per-window basis. The mode selected is a function of the depth of the window created and double-buffer mode.

In 8-bit mode, each pixel is used as an index into a 256-entry color map. Each entry in the color map provides eight bits per color for each of the red, green, and blue components, providing a color palette of over 16 million colors. Double-buffering is achieved by switching between two banks of 8-bit indexes. You can perform 3:3:2 direct color emulation in this mode.

In 12-bit mode, each pixel is represented by four bits each for red, green, and blue.

In 24-bit mode, a pixel is represented by eight bits each for red, green, and blue.

There are four hardware color maps available for use with the image planes. All four color maps are shared by all graphics processes. The information in this chapter provides more specific details on using these color maps.

In addition to the four hardware color maps in image planes, there are two hardware color maps for the overlay planes. One of the hardware color maps has the default X11 color map permanently installed in it. This is done to avoid technicolor in X11 and HP CDE windows and backgrounds.

The X server works only in combined mode. For information on supported X server modes, read the section "Supported X Server Modes" in the *Graphics Administration Guide*.

The HP VISUALIZE-FX family's device driver dithers all vectors, when dithering is enabled. This is true even when Color Recovery is enabled.

HP Visualize-FX^2 Device Description

The HP VISUALIZE-FX^2 frame buffer includes 24 image planes and 8 overlay planes. The image-plane visuals that are supported by the HP VISUALIZE-FX^2 are 8/8 double-buffered, 8-plane single-buffered, 12/12 double-buffered, 12-plane single-buffered, and 24-plane single-buffered. Note that Starbase does not support the following visuals on any device:

- depth 4 PseudoColor visual, single- or double-buffered
- depth 12 PseudoColor visual

The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer.

HP Visualize-FX^4 and HP Visualize-FX^6 Device Descriptions

From an application developer's or user's point of view, the only difference between the HP VISUALIZE-FX^4 and the HP VISUALIZE-FX^6 is that the HP VISUALIZE-FX^6 has higher performance.

The HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 frame buffers include 48 image planes and 8 overlay planes. Visuals supported by the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 are 8/8 double-buffered, 8-plane single-buffered, 24/24 double-buffered, and 24-plane single-buffered. Visuals that are not supported by the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 are 12/12 double-buffered, 12-plane single-buffered, and 48-plane single-buffered. Note that Starbase does not support the following visuals on any device:

- depth 4 PseudoColor, single- or double-buffered
- depth 12 PseudoColor visual

The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer.

Performance Information

The following performance information is specific to the HP VISUALIZE-FX family of graphics devices. See the *Graphics Administration Guide* for device-general performance information. In cases where this performance information conflicts with the device-general information in the *Graphics Administration Guide*, this device-specific information should be considered correct for HP VISUALIZE-FX devices.

- Performance is fastest for unobscured windows, or windows obscured by overlay windows only. There is a slight performance degradation for obscured windows.
- The number of vertices in a polygon does not cause HP VISUALIZE-FX devices to leave the performance-optimized path.
- Z-buffering enabled or disabled does not impact performance on HP VISUALIZE-FX devices.

Geometry Accelerator

Each HP VISUALIZE-FX device includes, by default, at least one geometry accelerator to provide high performance 3D solids modeling and high performance 3D wireframe with anti-aliasing. The HP VISUALIZE-FX geometry accelerators have a dedicated 24-bit Z-buffer. The primary use of the HP VISUALIZE-FX geometry accelerators is for 3D solids modeling, including drawing Starbase polygons, polyhedrons, rectangles, triangle strips, quadrilateral meshes, spline surfaces, geometry transform, and lighting and shading of primitives. The geometry accelerators have capabilities for both surface rendering and volumetric rendering.

Note that the geometry accelerators are not used for rendering in the overlay planes.

The following lists provide information to help you maximize your application performance. The first list describes operations that yield the best performance on the HP VISUALIZE-FX devices:

- Isotropic modeling transformations
- Lighting, with no more than 8 lights of any type
- View clipping
- Perspective and orthographic (parallel) transformations
- Depth cueing
- 3- and 4-sided filled primitives, with or without RGB, alpha, and normal data per vertex
- Triangle strips, with or without RGB, alpha, and normal data per vertex
- 2D and 3D polylines
- Wide lines

The following features use the HP VISUALIZE-FX geometry accelerators, but yield somewhat lower performance than the base features listed above.

- Non-convex polygons with more than 4 vertices
- Polyhedrons with move/draw flags
- Facet normal lighting
- Facet color

The following features bypass the HP VISUALIZE-FX geometry accelerators and use PowerShade instead:

- Self-intersecting polygons
- Model clipping/capping
- Deformation
- Backface distinguishing (but not back-face culling)
- Starbase `INT_OUTLINE` interior style
- Circles, ellipses, arcs
- Picking
- `move3d()/draw3d()`
- Polymarkers
- Rectangles
- Text

Note that the geometry accelerator directly handles polygons with 3 or 4 vertices only; more complicated polygons are decomposed into triangles. Convex polygons will be decomposed most easily. Non-convex polygons set will be decomposed less easily. Polyhedrons with move/draw flags will be decomposed, but with a significant penalty in execution time. Self-intersecting polygons can not be decomposed by the geometry accelerator. Instead, they are lit, shaded, and transformed by PowerShade, with only the final rendering steps performed by the HP VISUALIZE-FX scan conversion hardware. Nonplanar polygons or polygons with greatly differing colors or normals at the vertices will vary more than planar polygons or polygons with more homogeneous vertex data. For example, a quadrilateral with vertices that are slightly different shades of green is decomposed more easily than the same quadrilateral with one green, one blue, one red, and one black vertex.

Also, note that compound primitives (triangle strips, quadrilateral meshes, and polyhedrons) will perform better than the equivalent multiple discrete polygon calls, since the shared library call overhead is less.

For more information about specific primitives and their relative speeds, see the *Graphics Administration Guide*.

Texture Mapping Accelerator

An optional accelerator for texture mapped primitives may be purchased for use with the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 hardware. From an application developer's or user's point of view, the only difference between these texture mapping accelerators is that HP VISUALIZE-FX^6 has better texture mapping performance than HP VISUALIZE-FX^4. HP VISUALIZE-FX^2 does not support a texture mapping accelerator. Use the `graphinfo` program to determine whether your system has this optional accelerator. The line:

```
texture accelerator:        yes
```

will be present if and only if the texture accelerator hardware is installed. If the texture accelerator hardware is not installed, the line will read:

```
texture accelerator:        no
```

On supported devices, this hardware accelerates the following texture mapping features:

- Single texture map per primitive
- Full MIP mapping with all MIP interpolation filters
- All post-lighting texturing and pre-light replace and modulate texturing

The accelerator has memory built into it to hold up to 16 megabytes of texture data (with 8-bits red, green, blue, and alpha data per texel). This is enough memory for three 1024×1024 fully MIP-mapped textures, or a single 2048×2048 point-sampled texture map. However, through a caching scheme for the hardware texture memory, textures as large as 32768×32768 may be accelerated. Note that up to 4096 textures of size 64×64 or smaller, or 256 textures of size 256×256 or larger can be supported at one time.

To support the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 texture cache, a texture interrupt management daemon runs continuously. This daemon, named `timd`, is responsible for ensuring that the appropriate sections of texture maps reside in the hardware texture memory. As with other system processes, do not attempt to kill `timd`, as this may cause the hardware to enter a "hung" state from which it is difficult to recover.

Overlay Plane Rendering

Either the `hpvmx` or `sox11` device driver is used for Starbase rendering to the overlay planes. For more information on these device drivers, see the chapters "HP Virtual Memory and X" in this Addendum and "The Starbase-on-X11 Device Driver" in the *HP-UX Starbase Device Drivers Manual*.

If an overlay plane window is `gopen`ed with a driver name of `NULL`, the `hpvmx` driver will be used. See the table, "Driver Selection at gopen" in the chapter "HP Virtual Memory and X" in this Addendum for details.

8/8 VM double-buffering is supported in the overlay planes using the `hpvmx` driver.

HP Color Recovery Technology

The HP VISUALIZE-FX devices use HP Color Recovery for shaded fill areas in depth 8 image-plane visuals (for example, polygons and spline surfaces). Color Recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. HP Color Recovery is available in all depth 8 visuals on the HP VISUALIZE-FX graphics devices.

There are two components to HP Color Recovery. A different dither cell size (16×2) is used when rendering shaded polygons, and a digital filter is used when displaying the contents of the frame buffer to the screen.

The HP VISUALIZE-FX devices provide HP Color Recovery whenever you are in `CMAP_FULL` mode and you have used the `INIT` flag in the `gopen`, `shade_mode`, or the `double_buffer` function to initialize color maps. Keep in mind that the default color map mode is `CMAP_NORMAL` for PseudoColor visuals. Therefore, the HP Color Recovery color map will not be downloaded until you call `shade_mode` to set the mode to `CMAP_FULL` and use `INIT`.

HP Color Recovery is available when using either PseudoColor or TrueColor visuals. The HP Color Recovery color map is a read-only color map. Any attempts to change it will be ignored and no error will be reported.

In `CMAP_FULL` shade mode, disabling HP Color Recovery results in normal dithering of shaded fill areas. HP Color Recovery is not available with any other shade mode.

HP Color Recovery is enabled in conjunction with a particular X color map that is associated with your window. If that X color map is not currently installed in hardware by your window manager, you will not see the effect of the HP Color Recovery filter.

Note that vectors are always dithered, even in an HP Color Recovery window.

Under some conditions HP Color Recovery can produce undesirable artifacts in the image. This also happens with 4×4 dithering, but the artifacts are different. However, images rendered with HP Color Recovery are seldom worse than what dithering produces, and in most cases, HP Color Recovery produces significantly better pictures than dithering. Note that 4×4 dithering, like HP Color Recovery, is available in the `CMAP_FULL` color map mode, but *not* in the `CMAP_NORMAL` color map mode.

HP Color Recovery is available by default. If you wish to disable HP Color Recovery, you can do it in one of three ways:

- Add the screen option `DisableColorRecovery` to your `X*screens` file. Setting this screen option prior to starting up the X server disables HP Color Recovery for *all* applications and any attempts to enable HP Color Recovery will be ignored. Remember, if you set this screen option prior to starting up the X server, you cannot re-enable HP Color Recovery from the command line or from within an application. To set this screen option, add the following lines to your ⟨*x11-admin*⟩[1]`/X*screens` file:

  ```
  ScreenOptions
     DisableColorRecovery
  ```

  and restart HP CDE or X11. To restart HP CDE, log out, then select "Reset Login Screen" from the "Options" pull-down on the HP CDE log-in window.

- Export the environment variable `HP_DISABLE_COLOR_RECOVERY` before running your application. Setting this environment variable to any value disables HP Color Recovery for subsequently executed applications. To set this environment variable in your current X11 window, execute this command on the command line before running your application (assuming you are using the Korn shell):

  ```
  export HP_DISABLE_COLOR_RECOVERY=TRUE
  ```

[1] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

- Disable HP Color Recovery programmatically by using the Starbase `gescape` `COLOR_RECOVERY_CONTROL`. For details on this `gescape`, read the subsequent section "Gescapes."

Gescapes

The `COLOR_RECOVERY_CONTROL` `gescape` can be used to disable HP Color Recovery. Passing it a 0 value in `arg1` will disable HP Color Recovery, a 1 value will enable it (HP Color Recovery is enabled by default). The `arg2` parameter is ignored. The effect of this `gescape` will not take place until the next time you call `shade_mode` or `double_buffer` with the `INIT` flag. For example:

```
gescape_arg  arg1;
/* Disable HP Color Recovery */
arg1.i[0] = 0;
gescape(fildes, COLOR_RECOVERY_CONTROL, &arg1, NULL);
shade_mode(fildes, CMAP_FULL|INIT,0);
```

The Frame Buffer

Physical Address Space

The physical frame buffer is addressed as 2048×1024 bytes. The last 768 bytes of each line of the address space (those to the right of the screen) are not displayed and no memory exists in those areas.

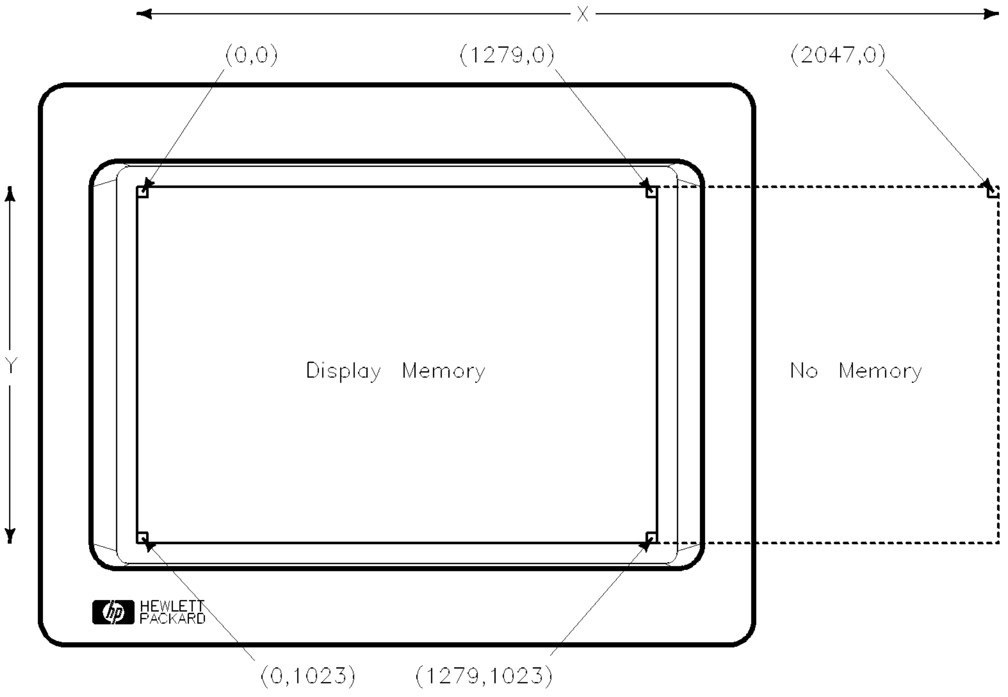

Figure 4-1. Physical Address Space

To Access the Frame Buffer Directly

When using the `R_GET_FRAME_BUFFER` gescape for direct user access to the frame buffer, correct access can only be assured by using the `R_LOCK_DEVICE` and `R_UNLOCK_DEVICE` gescapes.

1. Use `R_LOCK_DEVICE` just prior to direct frame buffer access.
2. Use `R_UNLOCK_DEVICE` directly after the frame buffer access and before any other Starbase commands.

Caution Do not read from or write to the offscreen addresses. Such operations will cause errors.

Frame Buffer Address Mapping

The frame buffer is organized as a single one-dimensional array of pixel values. The first byte (byte 0) of the frame buffer represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1279 is the last (right-most) displayable pixel on the top line. The next 768 bytes are not displayable. Byte 2048 is the first (left-most) pixel on the second line from the top. The last (lower right corner) pixel on the screen is byte number 2,096,383 ($1023 \times 2048 + 1279$).

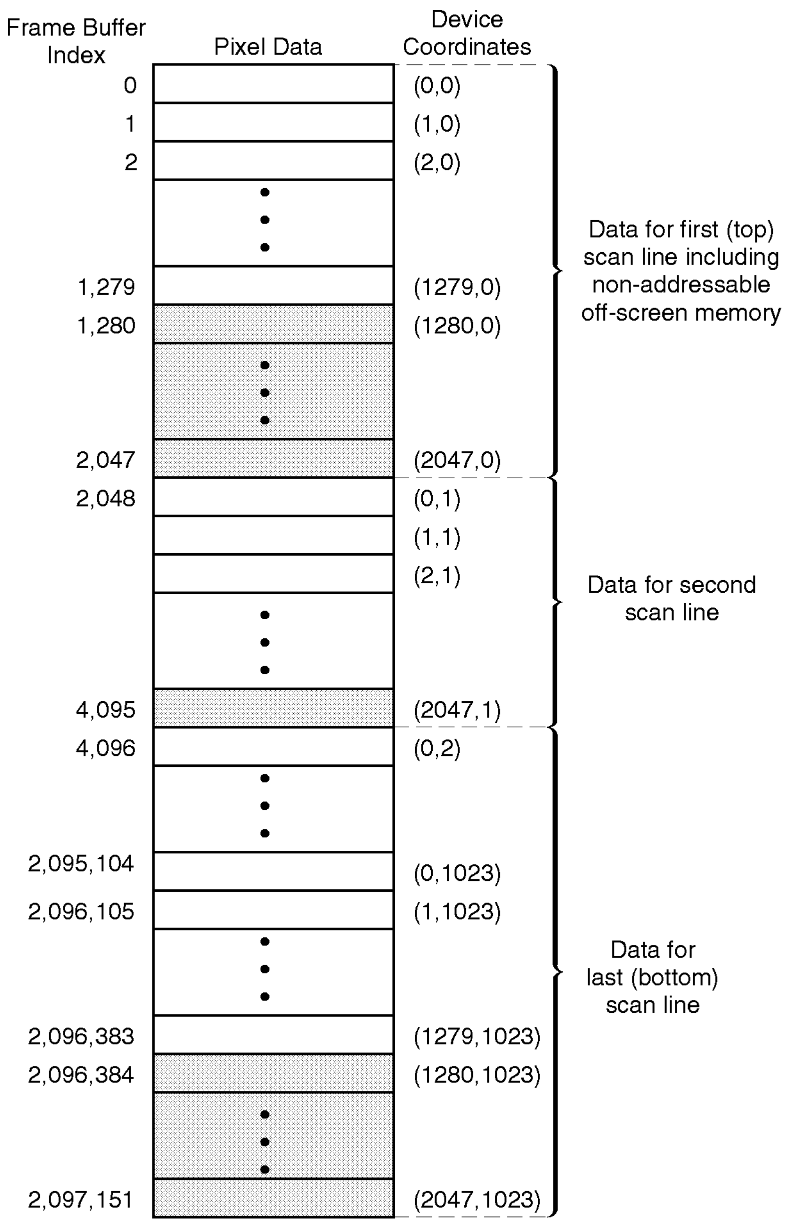

Figure 4-2. Frame Buffer Mapping in Memory

The HP VISUALIZE-FX^2 frame buffer has three banks of 8 planes each. The HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 frame buffers have six banks of 8 planes (two for each color). Only one bank can be accessed at a time. Use the `bank_switch` call to select a bank to read or write data directly from the frame buffer. For `block_read` and `block_write` operations to the image planes, the data is in all eight bits of each byte.

The default for reading the Z-buffer is always 24 bits per pixel in a 32-bit word. The Z-buffers for the HP VISUALIZE-FX devices are 23 bits deep, and their Z-buffer data is left justified in the lower 24 bits of the 32-bit word (that is, the 23-bit Z-buffer data is shifted left one bit from the least-significant bit), as shown in the following figure.

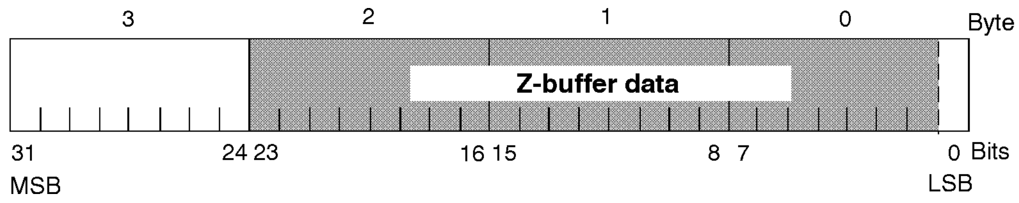

Figure 4-3. Hardware Z-Buffer Data Alignment

The `raw` parameter to `block_read` and `block_write` must be set to true in order to read from or write to the Z-buffer. Using `wbank=3` in the `bank_switch` command on the HP VISUALIZE-FX^2 selects the Z-buffer for reads or writes. Using `wbank=6` in the `bank_switch` command on the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 selects the Z-buffer for reads or writes.

The 24 bits of the Z-buffer reside in the lower 24 bits of the word that is used by `block_read` and `block_write`.

Unlike the frame buffer, the Z-buffer data is contiguous. The HP VISUALIZE-FX device's Z-buffers are always 1280×1024 where word 1280 is the leftmost word of the second scanline. For the HP VISUALIZE-FX family of devices, the Z-buffer is the size of the window. For example, if the window is 400×400, word 400 is the leftmost Z-buffer value for the second scan line.

Frame Buffer Configurations

The following table shows which color map modes are supported for different frame buffer configurations.

Table 4-2. Supported Frame Buffer Configurations

Device	Number of Planes	Supported Configurations
HP VISUALIZE-FX[2], HP VISUALIZE-FX[4], HP VISUALIZE-FX[6]	8, 8/8	`CMAP_NORMAL`, `CMAP_FULL`, `CMAP_MONOTONIC`
HP VISUALIZE-FX[2]	12, 12/12, 24	`CMAP_FULL`
HP VISUALIZE-FX[4], HP VISUALIZE-FX[6]	24, 24/24	`CMAP_FULL`

Since Starbase supports double-buffering per window, it is better to request double-buffering with a depth of 24 when in `CMAP_FULL` mode on HP VISUALIZE-FX[4] and HP VISUALIZE-FX[6] devices, or with a depth of 12 when in `CMAP_FULL` mode on a HP VISUALIZE-FX[2]. Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported in depth 8 windows for compatibility with previous devices, however, it is not recommended. Note that the 4/4 double-buffering mentioned here does not use a depth 4 visual. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes. Video tearing may occur, however, as this kind of double-buffering is not synchronized to the video refresh.

Using Starbase in X Windows

This section contains device specific information needed to run Starbase programs in X11 windows. If you need a general, device-independent explanation of using Starbase in X11 windows, refer to the "Using Starbase with the X Window System" chapter of *Starbase Graphics Techniques*.

To reduce the complexity of having multiple X server modes, the HP VISUALIZE-FX devices only support one X server mode. Several other key features have been designed to improve the overall usability of the devices in the X11 windows environment, and to reduce interaction issues between the X11 user interface and graphics library APIs (such as Starbase), that provide *direct hardware access* (DHA).

Per-Window Double-Buffering

The HP VISUALIZE-FX devices support double-buffering in the images planes on a per-window basis. See the table in the "Frame Buffer Configurations" section for information on configurations that support double-buffering in the image planes. All HP VISUALIZE-FX devices support 8/8 double-buffering in the overlay planes for each of the Starbase color map modes (`CMAP_FULL`, `CMAP_NORMAL`, and `CMAP_MONOTONIC`) in software. Remember that hardware double-buffering is not supported in the overlay planes.

Note that Starbase uses the `hpvmx` device driver to perform double-buffering in software in the overlay planes. This double-buffering method is slower than the hardware double-buffering used in the image planes.

Available Color Map Entries

The HP VISUALIZE-FX graphics devices have two hardware color maps in the overlay planes and four hardware color maps in the image planes.

If you query the X server for the number of entries in the default overlay visual's color map while you are using the default X server mode of the HP VISUALIZE-FX devices, the server will reply that there are 256 entries available. Although all 256 entries are available for use by an application, the last entry (index 255) is not writable because it is allocated by the X server.

Starbase Color Maps and X11 Read/Write Restrictions

The X color model defines read/write restrictions both on color maps and on individual entries in color maps. Starbase does not overwrite read-only color maps or color map entries as defined in the X color model. Attempts to write to color map entries in read-only color maps (that is, for TrueColor, StaticColor, or StaticGray visuals) are silently ignored.

Accessing HP Color Recovery with X Windows

The HP VISUALIZE-FX devices support HP Color Recovery for shaded areas. When a depth 8 window is used, HP Color Recovery will generate a better picture by attempting to eliminate the graininess caused by dithering. Color Recovery is available on all depth 8 windows on HP VISUALIZE-FX devices. For more information about HP Color Recovery, read the section "HP Color Recovery" found in this chapter.

The Starbase, HP PEX, and HP-PHIGS graphics libraries provide programmers who use these APIs with transparent access to the HP Color Recovery capability of the HP VISUALIZE-FX devices. If you are producing graphics using Xlib calls, then your application must perform some of the necessary processing. At server start-up, there is one property that is defined and placed on the root window if the `HP_DISABLE_COLOR_RECOVERY` environment variable has *not* been exported. This property is:

```
_HP_RGB_SMOOTH_MAP_LIST
```

The above property is of type `RGB_COLOR_MAP` and carries pointers to structures of type `XStandardColormap`. It may be interrogated with calls to `XGetRGB-Colormaps`. The property `_HP_RGB_SMOOTH_MAP_LIST` is a list of color maps that are associated with visual IDs that support HP Color Recovery. When the `XGetRGBColormaps` routine searches throughout this list for a color map with a visual ID that matches your window's visual ID and it finds one, your application knows that your visual supports HP Color Recovery, and uses that color map for any HP Color Recovery window.

HP Color Recovery uses all 256 entries of one of the available color maps. The color visual used by HP Color Recovery emulates the 24-bit TrueColor visual. Thus, the colors red, green, and blue are typically declared as integers in the range from 0 to 255. Note that each window that uses HP Color Recovery will use the same color map.

For HP Color Recovery to produce the best results, the emulated 24-bit TrueColor data is dithered as explained below.

A pixel to be dithered is sent to the routine provided in this example. Note that the values of the variables `RedValue`, `GreenValue` and `BlueValue` are generated by an application. In this example, the color values are assumed to be in the range [0..255].

The given routine receives the color values and the X and Y window address (`Xp` and `Yp`) of the pixel. The X and Y address is used to access the dither tables. The values from the dither tables are added to the color values. After the dither addition, the resultant color values are quantized to 3 bits of red and green and 2 bits of blue. The quantized results are packed into an 8-bit `unsigned char` and then stored in the frame buffer. As the contents of the frame buffer are scanned to the CRT, a special section in the HP VISUALIZE-FX device hardware then converts the 8-bit data stored in the frame buffer into a 24-bit TrueColor image for display.

Here is a routine that can be used to dither the 24-bit TrueColor data.

```
unsigned char dither_pixel_for_CR(RedValue,GreenValue,BlueValue,Xp,Yp)
int RedValue,GreenValueBlueValue,Xp,Yp;
{
static short dither_red[2][16] = {
  {-16,  4, -1, 11,-14,  6, -3,  9,-15,  5, -2, 10,-13,  7, -4,  8},
  { 15, -5,  0,-12, 13, -7,  2,-10, 14, -6,  1,-11, 12, -8,  3, -9} };

static short dither_green[2][16] = {
  { 11,-15,  7, -3,  8,-14,  4, -2, 10,-16,  6, -4,  9,-13,  5, -1},
  {-12, 14, -8,  2, -9, 13, -5,  1,-11, 15, -7,  3,-10, 12, -6,  0} };

static short dither_blue[2][16] = {
  { -3,  9,-13,  7, -1, 11,-15,  5, -4,  8,-14,  6, -2, 10,-16,  4},
  {  2,-10, 12, -8,  0,-12, 14, -6,  3, -9, 13, -7,  1,-11, 15, -5} };

int red, green, blue;
int x_dither_table, y_dither_table;
unsigned char pixel;

x_dither_table = Xp % 16;   /* X Pixel Address MOD 16 */
y_dither_table = Yp % 2;    /* Y Pixel Address MOD 2 */

red = RedValue;
green = GreenValue;
blue = BlueValue;

if (red >= 48) /* 48 is a constant required by this routine */
   red=red-16;
else
   red=red/2+8;
red += dither_red[y_dither_table][x_dither_table];
if (red > 0xff) red = 0xff;
if (red < 0x00) red = 0x00;

if (green >= 48) /* 48 is a constant required by this routine */
   green=green-16;
else
   green=green/2+8;
green += dither_green[y_dither_table][x_dither_table];
if (green > 0xff) green = 0xff;
if (green < 0x00) green = 0x00;

if (blue >= 112) /* 112 is a constant required by this routine */
   blue=blue-32;
else
   blue=blue/2+24;
blue += (dither_blue[y_dither_table][x_dither_table]<<1);
if (blue > 0xff) blue = 0xff;
if (blue < 0x00) blue = 0x00;

pixel = ((red & 0xE0) | ((green & 0xE0) >> 3) | ((blue & 0xC0) >> 6));

return(pixel);
}
```

Backing Store

Backing store is not supported by the `hpvisx` device driver. To use backing store on HP VISUALIZE-FX devices, you must use the `hpvmx` device driver. For image plane windows, you need to detect window exposure events and repaint the window when a previously obscured portion of a window is made visible.

X11 Cursor

The X11 cursor (also called the sprite) is maintained by the display hardware and never interferes with the frame buffer contents in either the image or overlay planes.

Supported Visuals

The following table of supported visuals contains information for programmers using either Xlib graphics or Starbase. The table lists the image plane depths of windows and color map access modes that are supported for a given graphics device. It also indicates whether or not backing store (also known as "retained raster") is available for a given visual, and lists the double-buffer configurations supported by Starbase for this device driver.

Table 4-3. Supported Visuals

Device	Depth	Visual Class	Backing Store		Starbase Double-Buffer[1]
			Xlib	Starbase	
HP VISUALIZE-FX[2], HP VISUALIZE-FX[4], HP VISUALIZE-FX[6]	8	PseudoColor TrueColor	Yes[2] Yes[2]	No[3] No[3]	8, 8/8 8, 8/8
HP VISUALIZE-FX[2]	12	DirectColor TrueColor	Yes Yes	No No	12, 12/12 12, 12/12
HP VISUALIZE-FX[2]	24	DirectColor TrueColor	Yes Yes	No No	24 24
HP VISUALIZE-FX[4], HP VISUALIZE-FX[6]	24	DirectColor TrueColor	Yes Yes	No No	24, 24/24 24, 24/24

1 Double-buffering with less than 8 planes (4/4, 3/3, 2/2, 1/1) is supported for compatibility with previous devices, however, it is not recommended. The `write_enable` and `display_enable` masks are used to accomplish double-buffering with less than 8 planes in a depth 8 visual. Flashing may occur, however, as this kind of double-buffering cannot be done on a per window basis. Note that double-buffering with less than 8-planes is *only* supported in `CMAP_NORMAL`.

2 Xlib primitives are supported by backing store. Whenever backing store is not maintained, normal expose events are generated.

3 Backing store is only supported when rendering with the `hpvmx` driver.

Moving the Default Visual to the Image Planes

Note By default the overlay planes have the default X11 color map permanently locked into one hardware color map, and any other hardware color maps used by the overlay planes are available for applications to use. Moving the default visual into the image planes will limit the number of hardware color maps available to you. In this mode, the HP VISUALIZE-FX devices provide a single hardware color map in the overlay planes.

X Windows provides a method for changing the default visual from a depth 8 overlay PseudoColor visual to a depth 8 image PseudoColor visual. This is done by moving the default visual to the depth 8 image PseudoColor visual. To do this, use SAM. Or to manually change the location of the default visual, edit the file[2]:

⟨*x11-admin*⟩`/X*screens`

and add the following lines:

```
Screen ⟨dev⟩/crt
   DefaultVisual
      Class PseudoColor
      Depth 8
      Layer Image
```

The `*` in the `X*screens` file name specifies the *display_number*. To determine the display number, execute this shell command:

```
echo $DISPLAY
```

Your results will have the following syntax:

⟨*host_name*⟩`:`⟨*display_number*⟩`.`⟨*screen_number*⟩

[2] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Here is an example of what your display name might look like after executing the `echo $DISPLAY` shell command:

```
mysystem:0.0
```

where *host_name* is `mysystem`, *display_number* is 0, and *screen_number* is 0. In the above example, you would edit the file:

⟨*x11-admin*⟩`/X*screens`

Note that the syntax of this specification has changed. For more information, see the file:

⟨*x11*⟩`/Xserver/info/screens/hp`

Device Support for the TrueColor Visual

TrueColor Visual Description

A TrueColor visual can be thought of as having a read-only color map where, for any given pixel value, about one third of the bits are used to describe each of the red, green, and blue colors, respectively. For an 8-plane TrueColor visual, 3 bits describe the red component, 3 bits describe the green component, and 2 bits describe the blue component. A 12-plane TrueColor visual uses 4 bits each to describe the red, green, and blue components. A 24-plane TrueColor visual uses 8 bits each to describe the red, green, and blue components. This is illustrated as follows:

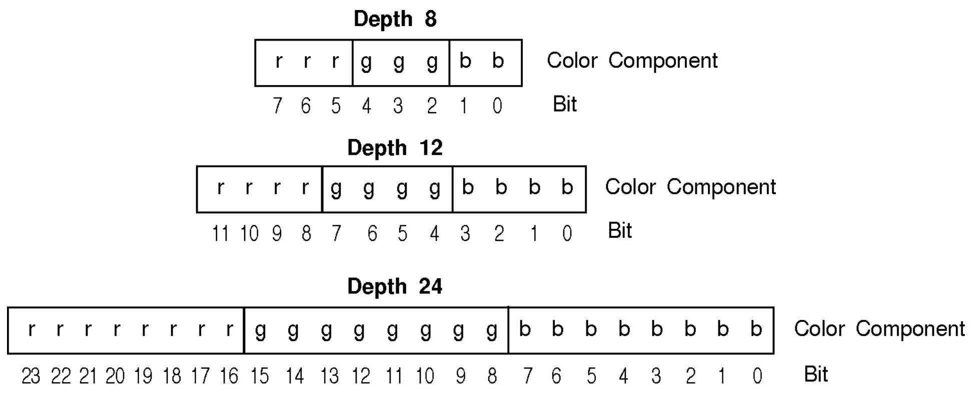

Figure 4-4. Pixel Representation for the Depth 8, 12, and 24 TrueColor Visuals

The following example refers to a depth 8 TrueColor visual; however, the example can be expanded to apply to depth 12 and 24 plane TrueColor visuals.

Example

Since the red and green components consist of 3 bits each, there are 8 different shades of red and 8 different shades of green available. There are 4 different shades of blue represented by 2 bits. As the component value increases, the intensity of that color increases. For example, a red component of 000 represents no red and a red component of 111 represents full red. Therefore, pixel value 0 is 000 red, 000 green, and 00 blue, which results in black. Pixel value 255 is 111 red, 111 green, and 11 blue, which results in white. These and other examples are shown in Table 4-4.

Table 4-4. Examples of Pixel Color Values

Pixel Value	Binary	Red	Green	Blue
0	000 000 00	shade 0	shade 0	shade 0
53	001 101 01	shade 1	shade 5	shade 1
139	100 010 11	shade 4	shade 2	shade 3
218	110 110 10	shade 6	shade 6	shade 2
255	111 111 11	shade 7	shade 7	shade 3

Note that the red, green and blue intensities for the color shades increase uniformly between 0 and 255.

Device Specific Visuals Information

The TrueColor Visual always uses a `shade_mode` of `CMAP_FULL`. When the TrueColor visual window is `gopen`ed, your application will automatically be in `CMAP_FULL` mode and the `shade_mode` call will ignore any attempts to go into another mode.

With the addition of the TrueColor visual at the 9.03 release of HP-UX, you need to consider the following information:

- If you originally created your windows using a command line similar to the following:

  ```
  xwcreate -g 600x500 -depth 24 window
  ```

  you will have to change how you create your windows by using command lines similar to this:

  ```
  xwcreate -g 600x500 -depth 24 -visual TrueColor window
  xwcreate -g 600x500 -depth 24 -visual DirectColor window
  ```

  Note the addition of the command line option `-visual` for declaring TrueColor and DirectColor visuals. For HP VISUALIZE-FX devices, Starbase uses a default color map of PseudoColor for depth 8 windows, DirectColor for depth 12 windows, and DirectColor for depth 24 windows if the visual is not specified.

- The TrueColor color map is read-only, so it cannot be modified. Note that any attempt to modify the TrueColor color map will not produce an error message.

  One class of applications that could be affected by this are those that perform their own gamma correction.

- If your application searches for a visual by traversing the visual list returned by the X server, you will find that the order of visuals in this list has changed compared to pre-HP-UX 9.03 systems because of the addition of new visuals. Therefore, your application code should always explicitly search for a particular visual rather than assuming that it occurs in a fixed position within the list of visuals returned by X11.

The Overlay Plane Color Map Management Scheme

Many applications use the default X11 color map. A technicolor effect (color flashing) in the windows using the default color map occurs when a non-default color map is downloaded into the hardware color map that had previously contained the default X11 color map.

Because so many applications use the default X11 color map, and because the HP VISUALIZE-FX devices have two hardware color maps in the overlay planes, the behavior on these devices is to dedicate (that is, lock) one overlay hardware color map to always hold the default X11 color map. This means that the assigned default overlay hardware color map cannot have another color map downloaded to it. The other overlay hardware color map is available to applications that use color maps other than the default.

The following section describes the default frame buffer configuration for the HP VISUALIZE-FX devices.

- The default visual configuration is:
  - overlay planes
  - depth 8
  - PseudoColor
  - opaque (no transparency)

  with 256 color map entries. Note that when using the default X server mode of the HP VISUALIZE-FX devices, if you query the X server for the number of entries in the default color map, the server will reply that there are 256 entries available. Although these entries are available, the X server reserves the last entry (index 255). So, that entry is not writable and should not be used.

- The default X11 color map is locked into one of the hardware color maps in the overlay planes. For a description of how to move the default visual to images planes, read the section “Moving the Default Visual to the Images Planes” in this chapter.

Overlay Plane Transparency and the X Windows System

The default X11 mode on the HP VISUALIZE-FX devices does not provide an overlay visual with a transparent property. If you need an overlay color map that supports transparency, create the color map using the visual that has transparency in its `SERVER_OVERLAY_VISUALS` property (see the next section).

An overlay visual's transparency feature enables you to render opaque objects (for example, menus and text) to a transparent overlay window and at the same time view rendered objects in an image window. For example, you may want to show a map of a country without all of its internal borders, and then add the internal borders as you need them. This can be done by creating two X windows with the same geometry: one in the overlay planes and one in the images planes. The country's terrain and boundaries would be drawn in the image planes window and the internal borders in a transparent overlay window. For the best-looking application, either the image or overlay window should be created without a border in this example.

In the overlay visual that supports transparency, the number of overlay color map entries is always 255 because the last entry is the transparent color map value. If your application requires a count of 256 entries in your color map, you need to set the environment variable `CountTransparentInOverlayVisual`. To do this, use SAM. Or you can manually edit ⟨*x11-admin*⟩[3]`/X*screens` file to add the following option:

```
ScreenOptions
   CountTransparentInOverlayVisual
```

before starting the X11 server. Any attempts to modify entry 255 will have no effect on the color map. Note that this example applies to the overlay transparent visual, which is not the default visual for the HP VISUALIZE-FX devices.

[3] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

The Default Frame Buffer Configurations for the HP Visualize-FX Devices

Table 4-5 shows the X visuals that are supported by Starbase on the HP VISUALIZE-FX^2. This configuration is not changed by using overlay transparency.

Table 4-5. The Default Frame Buffer Configuration for HP Visualize-FX^2

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	2	no	PseudoColor[1]
overlay	8	single	2	yes	PseudoColor
image	8	single or double	4	N/A	PseudoColor[2] TrueColor
image	12	single or double	4	N/A	DirectColor TrueColor
image	24	single	4	N/A	DirectColor TrueColor

1 This is the default overlay visual.

2 This is the first visual returned by `xdpyinfo`.

Table 4-6 shows the X visuals that are supported by Starbase on the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6. These configurations are not changed by using overlay transparency.

Table 4-6.
The Default Frame Buffer Configuration for HP Visualize-FX^4 and HP Visualize-FX^6

Frame Buffer Layer	Window Depth	Hardware Buffering	Hardware Color Maps	Overlay Transparency	Visual
overlay	8	single	2	no	PseudoColor[1]
overlay	8	single	2	yes	PseudoColor
image	8	single or double	4	N/A	PseudoColor[2] TrueColor
image	24	single or double	4	N/A	DirectColor TrueColor

1 This is the default overlay visual.

2 This is the first visual returned by `xdpyinfo`.

If you need an overlay color map that supports transparency, create the color map using the visual that has transparency in its `SERVER_OVERLAY_VISUALS` property. To look at the contents of this property, you would use code similar to the following:

```
/* First, get the list of visuals for this screen. */
    .
    .
*pVisuals = XGetVisualInfo(display, mask, &getVisInfo, numVisuals);
    .
    .
/* Now, get the overlay visual information for this screen.  To obtain
 * this information, get the SERVER_OVERLAY_VISUALS property. */

overlayVisualsAtom = XInternAtom(display, "SERVER_OVERLAY_VISUALS", True);
if (overlayVisualsAtom != None)
{
    /* Since the Atom exists, we can request the property's contents. */
    bytesAfter = 0;
    numLongs = sizeof(OverlayVisualPropertyRec) / 4;
    do
```

```
        {
            numLongs += bytesAfter * 4;
            XGetWindowProperty(display, RootWindow(display, screen),
                               overlayVisualsAtom, 0, numLongs, False,
                               overlayVisualsAtom, &actualType, &actualFormat,
                               &numLongs, &bytesAfter, pOverlayVisuals);
        } while (bytesAfter > 0);
    }
        .
        .
    /* Process the pOverlayVisuals array. */
    while (--nVisuals >= 0) {
        nOVisuals = *numOverlayVisuals;
        pOVis = *pOverlayVisuals;
        imageVisual = True;
        while (--nOVisuals >= 0) {
            pOOldVis = (OverlayVisualPropertyRec *) pOVis;
            if (pVis->visualid == pOOldVis->visualID)
            {
                imageVisual = False;
                pOVis->pOverlayVisualInfo = pVis;
                /* Found the transparent visual */
                if (pOVis->transparentType == TransparentPixel);
            }
            pOVis++;
        }
    }
```

This program segment is not complete; however, its main purpose is to give you an idea of how a visual is checked for overlay transparency. The source for the above code can be found in the example source code for the Starbase Programming Environment[4]:

⟨*sb-utils*⟩/`wsutils.c`

[4] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

To Open and Initialize the Device for Output

Syntax Examples

C programs[5]**:**

```
fildes = gopen("⟨screen⟩/window", OUTDEV, NULL, INIT);
```

FORTRAN77 programs:

```
  fildes = gopen('⟨screen⟩/window'//char(0), OUTDEV,
+               char(0), INIT)
```

Pascal programs:

```
fildes := gopen('⟨screen⟩/window', OUTDEV, '', INIT);
```

Parameters for gopen

The `gopen` procedure has four parameters: *path*, *kind*, *driver*, and *mode*.

- *path* — This is the name of the device file created by `xwcreate`(1) or created with `XCreateWindow`(3X) and returned from `make_X11_gopen_string`(3G).
- *kind* — This parameter should be `OUTDEV` if the window will be used for output, `INDEV` if the window will be be used for Starbase input, or `OUTINDEV` if the window will be used for both output and Starbase input.
- *driver* — The character representation of the driver type. If this parameter is set to `NULL`, then `gopen` will inquire the device and use the appropriate driver.

  For example:

`NULL`	*for C.*
`char(0)`	*for FORTRAN 77.*
`''`	*for Pascal.*

[5] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Or, a character string may be used to specify a driver. For example:

`"hpvisx"`	*for C.*
`'hpvisx'//char(0)`	*for FORTRAN 77.*
`'hpvisx'`	*for Pascal.*

- *mode* — The mode control word consists of several flag bits *OR*ed together. Listed below are flag bits that have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure. See the *Starbase Graphics Techniques* manual for a description of `gopen` actions when accessing an X11 Window.
  - `0` (zero) — Open the device, but do nothing else. The software color table is initialized from the current state of the hardware color map.
  - `INIT` — Open and initialize the device as follows:
    1. The frame buffer is cleared to zeros.
    2. The color map is reset to its default values.
    3. The display is enabled for reading and writing.
    4. The Z-buffer is cleared.
  - `RESET_DEVICE` — Same as INIT.
  - `SPOOLED` — Not supported; raster devices cannot be spooled.
  - `MODEL_XFORM` — Opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
  - `INT_XFORM` — Perform only integer and common operations. All floating point operations will cause an error.
  - `INT_XFORM_32` — Perform only integer and common operations, with extended precision. All floating point operations will cause an error.

    This mode is provided for compatibility of integer precision with previous devices. `INT_XFORM` might use a faster transformation pipeline with slightly less precision. It is recommended to use `INT_XFORM` unless maximum precision is required. If maximum precision is required, even at the expense of performance, use `INT_XFORM_32`.
  - `FLOAT_XFORM` — Perform only floating point and common operations. All integer operations will cause an error.
  - `ACCELERATED` and `UNACCELERATED` — These flags are ignored by the HP VISUALIZE-FX devices. The `hpvisx` device driver is always accelerated, whether or not one of these flags is used.

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location (0, 0) is the upper-left corner of the window with X-axis values increasing to the right and Y-axis values increasing down.

Use this form of pixel addressing when calling high-level Starbase operations in terms of (`x,y`) device coordinates.

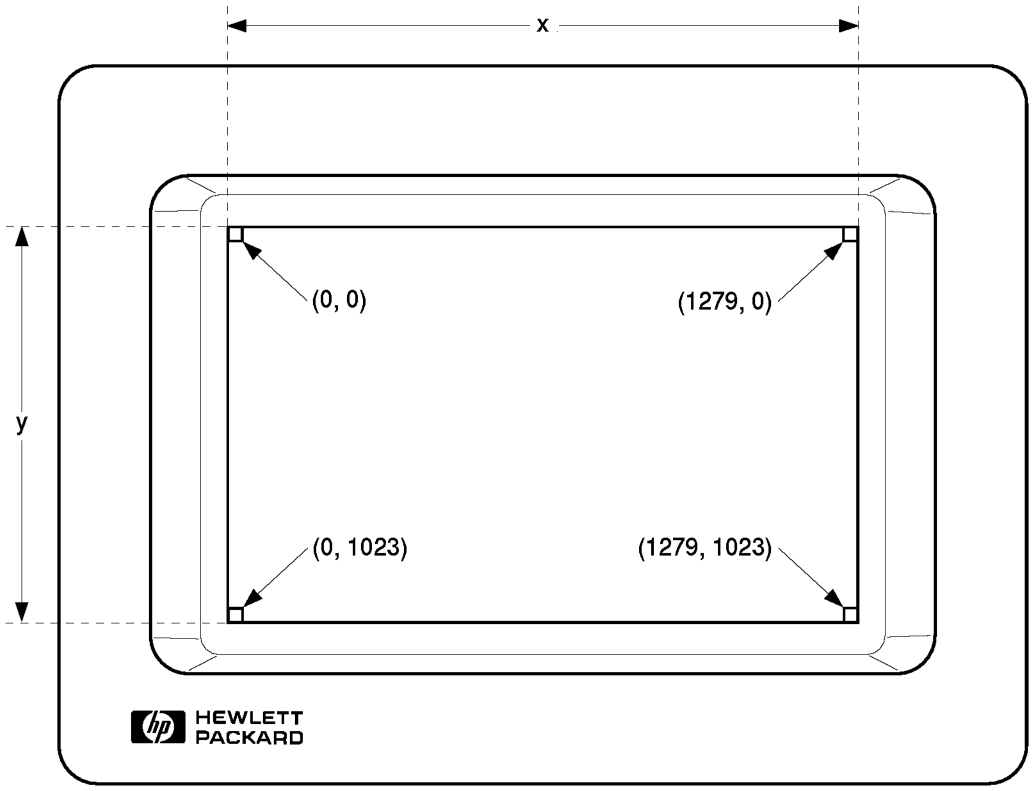

Figure 4-5. Device Coordinates

Starbase Echoes

This section provides information about the echo implementation for Starbase. The HP VISUALIZE-FX devices, and all future graphics device drivers will use this implementation of Starbase echoes.

Starbase echoes use Xlib functionality to draw echoes in the same planes as the visual that is active for the window. All previously supported Starbase echo functions are implemented except for those listed below:

- There is no support for the following gescapes:

  ```
  R_DEF_ECHO_TRANS
  R_ECHO_FG_BG_COLORS
  R_OV_ECHO_COLORS
  R_OVERLAY_ECHO
  ```

  In addition, the `R_ECHO_MASK` is only supported for a maximum of two colors within the raster definition.
- More than one Starbase echo per window is not supported.
- Rendering with both Xlib and Starbase in the same window while Starbase echoes are active may produce some random pixel “noise”.

Starbase Functionality

This section contains information on Starbase calls that are *not* supported by the HP VISUALIZE-FX devices. This section also contains information on `gescapes` that are supported by the HP VISUALIZE-FX devices.

Calls Not Supported on the HP Visualize-FX Devices

The following calls are not supported when using the HP VISUALIZE-FX devices:

```
bf_texture_index
contour_enable
define_contour_table
define_texture
texture_index
texture_viewport
texture_window
```

Conditional Support of Starbase Calls on the HP Visualize-FX Devices

The following calls are supported with the listed exceptions:

`alpha_transparency`	The HP VISUALIZE-FX devices support alpha transparency. Alpha only applies to filled areas such as polygons, quadrilateral meshes, triangular strips, and spline surfaces. Vector primitives are not rendered with `alpha_transparency`. The alpha transparency feature is limited to `CMAP_FULL` in a depth 12 or 24 visual. Only the floating point version of these primitives will be rendered with alpha transparency; device coordinate primitives do NOT use alpha. The HP VISUALIZE-FX devices do not support alpha transparency with attenuation. (See `alpha_transparency`(3G) in the *Starbase Reference* manual for the list of parameters). Note that Starbase does not utilize a hardware alpha buffer to implement alpha transparency.

`block_read`, `block_write`

The *raw* parameter for the `block_read` and `block_write` commands is used by this driver to do plane-major reads and writes. It is enabled by the gescape `R_BIT_MODE`.

The storage destination supplied by the user as the source or destination must be organized as follows.

- The data from each plane is packed with eight pixels per byte.
- Each row must begin on a byte boundary. Thus, the size of the rectangle as specified by the ⟨*length_x*⟩ and ⟨*length_y*⟩ parameters must correspond to an integral number of bytes.
- The data for the next plane begins on the following byte boundary.
- Clip to the screen limits.
- The first pixel in the source rectangle is placed in the high-order bit of the first byte in each plane region.
- When clipping, part of each plane region will not be read (`block_read`) or altered (`block_write`).

A bit mask selects the planes to read or write. The initial value of this mask is 1 (one) indicating that only plane 0 is to be accessed. The value of the mask may be changed using the `R_BIT_MASK` or `GR2D_PLANE_MASK` gescapes. `GR2D_PLANE_MASK` is discussed in the appendix of the *HP-UX Starbase Device Drivers Manual*. The planes selected by the mask are expected to reside in consecutive plane locations in the user storage area. This reduces the storage requirements to exactly what is needed but also presents the potential for addressing violations or undesirable results.

For example, if the plane mask is changed to specify more planes between a `block_read` and a following `block_write` from the same location, the `block_write` will attempt to access storage for planes that were not read (and perhaps not allocated). The application program must ensure consistency in these operations.

`fill_dither` — The ability to dither is disabled if the number of colors specified by `fill_dither` is one. However, if the number of colors specified is greater than 1, the default dither cell size of 16 is used. Dithering is only used in depth 8 visuals while in either the `CMAP_FULL` or `CMAP_MONOTONIC` color map mode, and in depth 12 visuals while in `CMAP_FULL` color map mode.

`interior_style` — The *style*s `INT_PATTERN` and `INT_HATCH` are not supported by the HP VISUALIZE-FX devices.

`light_source` — Up to 15 directional light sources are available on HP VISUALIZE-FX devices. The HP VISUALIZE-FX devices' hardware accelerates up to eight directional light sources. Using nine to fifteen directional light sources will cause a noticeable performance degradation.

`line_filter`, `perimeter_filter` — Anti-aliasing is supported on the HP VISUALIZE-FX devices. Anti-aliasing for this device applies only to floating point vectors. Device coordinate and integer (primitives that use `gopen`'s `INT_XFORM` *mode*) primitives do not use anti-aliasing. The anti-aliasing features are also limited to the `CMAP_FULL` color map mode in a depth 12 or 24 visual.

The HP VISUALIZE-FX devices have three anti-aliasing modes that may be specified with the `line_filter` and `perimeter_filter` procedures. The index values are assigned as follows:

`0` Anti-aliasing disabled, all vectors have one pixel wide output.

1 Anti-aliasing enabled, all vectors have two pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according the the formula:

$$\textit{pixel\_color} = (\textit{new\_pixel} \times \alpha) + (\textit{old\_pixel} \times (1 - \alpha));$$

2 Anti-aliasing enabled, all vectors have two pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according to the formula:

$$\textit{pixel\_color} = (\textit{new\_pixel} \times \alpha) + \textit{old\_pixel}$$

3 Anti-aliasing enabled, but with no alpha blending.

$$\textit{pixel\_color} = \textit{new\_pixel}$$

Note that this implementation of 2 pixel wide anti-aliasing differs from the CRX-48Z (with 3-wide anti-aliasing). It is also slightly different than the anti-aliasing implementation on the HP VISUALIZE-48 and HP VISUALIZE-48XP.

`shade_mode` The color map mode may be selected. Shading can be turned on only if using PowerShade. Shading is not supported on device coordinate primitives even with PowerShade. Note that the HP VISUALIZE-FX devices automatically use PowerShade.

`text_precision` Only `STROKE_TEXT` precision is supported.

`vertex_format` If using PowerShade software, `vertex_format` is fully functional. Note that the HP VISUALIZE-FX devices automatically use PowerShade.

`*_with_data`

The following routines are called `with_data` routines because they allow you to send extra vertex data. These `with_data` routines are supported by the HP VISUALIZE-FX devices.

```
partial_polygon_with_data3d
polygon_with_data3d
polyhedron_with_data
polyline_with_data3d
polymarker_with_data3d
polyquad_with_data3d
polytriangle_with_data3d
quadrilateral_mesh_with_data
triangle_strip_with_data
```

Note that the `TEXTURE_MAP` flag applies only to the TurboVRX devices via the `texture_*` routines. This extra data per vertex is not used in the `tm_*` routines.

For detailed information on these routines, read the *Starbase Reference* and "Appendix A" of this document. In some cases, you will be able to find the routines under their own name, but in other cases, you will need to use the first part of the routine name to locate these routines (e.g., `polyline_with_data3d` is described on the man page for `polyline(3G)`).

Supported Gescapes

The `hpvisx` device driver supports the following gescape operations on the HP VISUALIZE-FX configurations. Refer to Appendix A of the *HP-UX Starbase Device Drivers Manual* and Appendix A of this addenda for details on gescapes.

- `BLOCK_WRITE_SKIPCOUNT`—Specify byte skip count during block write.
- `COLOR_RECOVERY_CONTROL`—Disable HP Color Recovery.
- `CUBIC_POLYPOINT`—Specify points to be rendered in a cubic volume specified in modeling coordinates.
- `DC_PIXEL_WRITE`—Specify points to be rendered along a horizontal scan line.
- `DRAW_POINTS`—Select different modes of rounding for rendered points.
- `GAMMA_CORRECTION`—Enable/disable gamma correction.
- `GCRX_PIXEL_REPLICATE`—Pan and zoom a raster image.
- `IGNORE_RELEASE`—Trigger only when button pressed.
- `ILLUMINATION_ENABLE`—Turn on/off illumination bits.
- `LINEAR_POLYPOINT`—Specify points to be rendered along a line specified in modeling coordinates.
- `LS_OVERFLOW_CONTROL`—Set light source overflow handling.
- `POLYGON_TRANSPARENCY`—Segment control over front/back face screen.
- `READ_COLOR_MAP`—Read Color Map.
- `R_BIT_MASK`—Bit mask.
- `R_BIT_MODE`—Bit mode.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_LINE_TYPE`—User defined line style and repeat length.
- `R_LOCK_DEVICE`—Lock device.
- `R_READ_FB`—Write an image to a window whose shade mode is set to `CMAP_FULL`.
- `R_WRITE_FB`—Read the image out of a window created with the shade mode set to `CMAP_FULL`.
- `R_UNLOCK_DEVICE`—Unlock device.
- `SET_POLYGON_OFFSET`—Enable Z-buffer biasing of fill pixels.
- `STEREO`—Activate stereo output mode.
- `SWITCH_SEMAPHORE`—Semaphore Control.
- `TRANSPARENCY`—Set screen door transparency mask (front face and back face).
- `TRIGGER_ON_RELEASE`—Trigger only when button is released.
- `WIDELINE_CONTROL`—Turn on/off and set attributes of widelines.
- `ZBANK_ACCESS`—Enable/disable Z-buffer block operations.
- `ZWRITE_ENABLE`—Enable/disable replacement of Z value.

Exceptions to Gescape Support

Note Because gescape operations are device-dependent, the exceptions discussed below may be removed in future drivers.

`GAMMA_CORRECTION` Gamma correction is implemented by modifying the color map. It is available only in 12-bit visuals and 24-bit DirectColor visuals on the HP VISUALIZE-FX devices. For information on the gescape `GAMMA_CORRECTION`, refer to Appendix A in the *HP-UX Starbase Device Drivers Manual*. If a global gamma correction value has been set via the X server or the gamma correction tool, that global gamma correction value will be used and the color map will not be modified. See the *Graphics Administration Guide* for more information about the gamma correction tool.

The features involved (along with the names of the affected gescape operations) are listed below. For more details in the gescape operations, refer to Appendix A in the *HP-UX Starbase Device Drivers Manual*.

`R_BIT_MASK` The gescape operation `R_BIT_MASK` defines a plane mask to the driver that is used for bit/pixel access to a single plane in the frame buffer. As with other device drivers, only the plane corresponding to the highest bit set in the mask is transferred.

`R_BIT_MODE` When calling `block_read` or `block_write` with the *raw* parameter set to `TRUE`, the driver supports bit/pixel frame buffer access to single planes.

Modified Gescapes

The HP VISUALIZE-FX devices support the `Z_CLIP_VOXEL` flag for the `CUBIC_POLYPOINT` and `LINEAR_POLYPOINT` gescapes. This flag enables read-only Z-clipping of the point primitives against the existing contents of the Z-buffer. The Z-buffer is not modified.

For `CUBIC_POLYPOINT`, the `Z_CLIP_VOXEL` flag is one of several in the vertex format argument, `arg1.i[8]`. For `LINEAR_POLYPOINT`, the `Z_CLIP_VOXEL` flag is one of several in `arg1.i[8]`. Consult the gescape chapter for more information about `LINEAR_POLYPOINT` and `CUBIC_POLYPOINT`.

Comparison Between the HP Visualize-FX Family and Other Devices

Comparing the HP Visualize-FX^2 and HP VISUALIZE-24 Devices

HP VISUALIZE-FX^2 uses the `hpvisx` device driver instead of the `hphcrx` device drivers used by the HP VISUALIZE-24 graphics device.

The HP VISUALIZE-FX^2 graphics device is very similar to the HP VISUALIZE-24 graphics device. This allows applications written and delivered for the HP VISUALIZE-24 to use the HP VISUALIZE-FX^2 accelerator without requiring different executable code.

All the gescapes available for the HP VISUALIZE-24 are supported by the HP VISUALIZE-FX^2. Additionally, the `Z_BUFFER_COMPARE_RULE` gescape is supported on the HP VISUALIZE-FX^2 (but not the HP VISUALIZE-24).

HP VISUALIZE-FX^2 uses a different dithering algorithm than HP VISUALIZE-24. The HP VISUALIZE-FX devices implement a different dither matrix for each color channel, while the HP VISUALIZE-24 uses the same matrix for all three color channels. Patterns produced by dithering should be less noticeable on the HP VISUALIZE-FX^2 than the HP VISUALIZE-24.

HP VISUALIZE-FX^2 uses a new anti-aliasing filter, which will result in slightly different pixel colors than the HP VISUALIZE-24.

Possible behavioral differences between the HP VISUALIZE-FX^2 and HP VISUALIZE-24 devices are mostly because of hardware differences. These behavioral differences should not affect the operation of the application and may only be observed when directly comparing the images between the two devices. Some of these differences are discussed in the "Image Differences" section below.

Comparing the HP Visualize-FX^4/HP Visualize-FX^6 and HP VISUALIZE-48/HP VISUALIZE-48XP Devices

HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 use the `hpvisx` device driver instead of the `hphcrx48z` device driver used by the HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices.

HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 are very similar to the HP VISUALIZE-48 and HP VISUALIZE-48XP graphics devices. This allows applications written and delivered for the HP VISUALIZE-48 and HP VISUALIZE-48XP to use the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 accelerators without requiring different executable code.

All the gescapes supported on the HP VISUALIZE-48 and HP VISUALIZE-48XP are supported by the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6. Additionally, the `Z_BUFFER_COMPARE_RULE` gescape is supported on the HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 (but is not supported on the HP VISUALIZE-48 and HP VISUALIZE-48XP).

HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 use a different dithering algorithm than the HP VISUALIZE-48 and HP VISUALIZE-48XP. The HP VISUALIZE-FX devices implement a different dither matrix for each color channel, while the HP VISUALIZE-48/HP VISUALIZE-48XP use the same matrix for all three color channels. Patterns produced by dithering should be less noticeable on the HP VISUALIZE-FX^4/HP VISUALIZE-FX^6 than on the HP VISUALIZE-48/ HP VISUALIZE-48XP.

HP VISUALIZE-FX^4 and HP VISUALIZE-FX^6 use a new anti-aliasing filter, which will result in slightly different pixel colors than the HP VISUALIZE-48 and HP VISUALIZE-48XP.

Possible behavioral differences between the HP VISUALIZE-48/ HP VISUALIZE-48XP and the HP VISUALIZE-FX^4/HP VISUALIZE-FX^6 are mainly the result of hardware differences. These behavioral differences should not affect the operation of the application and may only be observed when directly comparing the images between the different devices. Some of these differences are discussed below.

Image Differences

Because of the different mechanisms used to generate the image, the images generated by the `hpvisx` and `hpvmx` device drivers will be different. The `ACCELERATED` and `UNACCELERATED` flags in `gopen` are ignored by the HP VISUALIZE-FX devices, which means Starbase will always use the `hpvisx` device driver in the image planes. The only exception to this is if the `hpvmx` device driver is specifically requested by `gopen`.

These minor image differences may also exist when comparing images between two different devices. For example, when comparing the HP VISUALIZE-FX^2 and HP VISUALIZE-24 devices.

These minor differences are listed below:

- Slight shifts in the image location on the display.
- Minor differences in color interpolation.
- Minor differences in pattern alignment or line type segment alignment.
- Specular highlights generated by lighting may differ slightly between the HP VISUALIZE-FX^2 and the HP VISUALIZE-24.

Note These differences should be minor. They may change in future releases of the Starbase graphics library.

`screenpr` for the HP Visualize-FX Devices

The *screenpr(1G)* command has been designed to use the X11 and the HP imaging library, rather than Starbase, to read to the screen that is displayed by a HP VISUALIZE-FX device. This implementation of *screenpr(1G)* correctly processes image and overlay planes, multiple color maps, and overlay transparency.

Note that command line options are still the same; however, if `screenpr` detects it is running on a HP VISUALIZE-FX device, it will use the `DISPLAY` environment variable to determine the screen to read, rather than using the device file path given by the `-F` option.

The `-p` option to print a single plane (and consequently the `-f` and `-b` options) is not supported on the HP VISUALIZE-FX versions of `screenpr`. These options will be ignored by `screenpr`.

This new version of `screenpr` uses X11 image library calls and executes *pcltrans(1G)* to produce PCL output. Therefore, `screenpr` may produce error messages from X, the HP imaging library, or `pcltrans`.

The HP VISUALIZE-FX `screenpr` implementation always expands the data to 24 bits. Therefore, the PCL output of an 8-bit only device will be approximately 3 times larger than might be expected.

5

HP Virtual Memory and X

Introduction

The `hpvmx` Starbase device driver, or HP VMX, offers application developers and end users an exciting and powerful tool for enhancing their Starbase graphics system usage. HP VMX allows you to use *any*[1] X11 graphics window (local or remote) as a "virtual device" for output of Starbase graphics.

In other words, HP VMX extends the X11 client/server model to include the 3D graphics functionality found in the Starbase graphics library. The HP VMX driver offers you the ability to run Starbase applications to all X11 servers to which you are able to run other X applications. Furthermore, most applications can take advantage of this extended capability with little or no modification.

This chapter covers information on HP VMX support, configuration, compiling/linking, and device characteristics. For a description of the HP VMX device driver and information on how to use it, read the section "Using the HP VMX Device Driver" in the chapter "Using Starbase with the X Window System" found in the *Starbase Graphics Techniques* manual. With this information, you will be able to determine how to utilize the capabilities of HP VMX.

Note HP VMX is supported on the Starbase, HP-PHIGS, and HP PEXlib graphics APIs.

[1] Refer to the "HP VMX Support" and "HP VMX Configurations" sections for details on official HP support of HP VMX.

Supported Visuals

HP-UX releases 10.20 and later offer support for both depth 8 and depth 24 visuals. The depth 24 visual support is new for HP-UX 10.20.

HP VMX and PowerShade

The 3D surfaces software, PowerShade, is fully supported on HP VMX. By combining PowerShade with HP VMX, you have the capability of rendering high performance 3D graphics including these features:

- Lighting and shading
- Hidden surface removal via 16-bit software Z-buffering
- Virtual memory double-buffering

These functions fully support the X11 client/server model.

HP VMX Support

There are two sides to the HP VMX support that must be separately addressed: HP VMX server support and HP VMX client support. The HP VMX server refers to the machine on which the Starbase application is executing (not necessarily being displayed) and the HP VMX client refers to the X server on which the Starbase images are being displayed.

For example, one supported configuration is to run a Starbase application using HP VMX on an HP 735 workstation across the network for display on an HP700/RX X Station. In this example, the HP 735 is the HP VMX server, and the HP700/RX X Station is the HP VMX client.

HP VMX Server Support

On the server side, HP VMX is supported on all HP Series 700 workstations running HP-UX 9.0 or later.

HP VMX Client Support

On the client side, HP VMX output may be directed to *any* depth 8 or depth 24 X11 window on your network, and is supported on all HP X11 servers, including:

- HP Series 700 Workstations running X11
- HP700/RX X Stations (X terminals)
- Other computers that can display a depth 8 or depth 24 X11 window.

HP VMX API Support

HP VMX is supported on the following graphics APIs:

- Starbase
- HP-PHIGS
- HP PEXlib.

For More Information

Information in this chapter is specific to HP VMX. For more information on backing store in X windows and linking shared or archived libraries, read the following manuals:

- See the *Starbase Graphics Techniques* manual for general Starbase programming information.
- Refer to the *Graphics Administration Guide* to read about linking shared or archive libraries, path naming conventions, X windows, completing installation, and setting up graphics devices.

Device Description

What is HP VMX?

In order to answer the question "What is HP VMX?" let us first examine its name. HP VMX is a shorthand name for the HP Virtual Memory X driver, and is derived from its implementation and usage. Briefly, HP VMX offers the capability to render 3D graphics images into *Virtual Memory* for display in the *X Window System* client/server environment.

While HP VMX is technically a Starbase "device driver" it differs somewhat from the traditional definition. A traditional Starbase device driver implements device-specific code necessary to support the device-independent Starbase graphics library on a particular graphics device (or family of graphics devices). HP VMX, on the other hand, implements the code to support the device-independent Starbase graphics library in an X11 graphics window — independent of the underlying hardware on which the X window resides.

Because HP VMX uses the X11 protocol to display the images, the targeted window may be local or remote, on HP or non-HP hardware, a workstation, an X terminal, or a Personal Computer[2]. The only requirement is that you output to an X11 graphics window. Note, too, that the application is not responsible for displaying the images via X11 protocol; this is handled by the HP VMX driver.

You may recognize similarities between HP VMX and the "Starbase-on-X11" (SOX11) device driver. While the X11-based client/server models are similar, differences do exist in both functionality (HP VMX has a richer set) and performance (differs per functionality). Please see the section "SOX11 vs. HP VMX" for an overview comparing and contrasting the two drivers.

[2] Refer to the "HP VMX Support" section for details on official HP support of HP VMX.

How Do You Use HP VMX?

The following example shows the steps necessary to run an application using HP VMX. This example is intended to give you a feel for the kinds of steps necessary to use HP VMX, rather than provide a detailed tutorial on all the steps necessary to explain each step. Refer to the sections throughout this chapter, including "Compiling Your Application with the HP-VMX Graphics Driver," and "To Open and Initialize the Device for Output" for details on these steps.

HP VMX Usage Example

In order to use HP VMX to run a PowerShade application to an HP700/RX X Station across the network from an HP 735 (running 9.0 HP-UX or later), you need to:

1. Purchase the "PowerShade for HP700/RX X Stations" license to allow you to run HP VMX to a remote X11 server.
2. Make sure PowerShade is installed on your server (the HP 735).
3. Execute an `xhost` command from your X Station to give the HP 735 (named `dspsvr` in this example) permission to access your X Station's local display server. For example,

   ```
   xhost +dspsvr
   ```

4. Create an `hpterm` window and `rlogin` to the HP 735 from your X Station.
5. Set the `DISPLAY` environment variable to the X Station's `DISPLAY` in this HP 735 `hpterm` window. Note that in the following example the X station is named `myxterm`. For example, in `ksh`:

   ```
   export DISPLAY=myxterm:0.0
   ```

6. Run your application from this window.

The application is now executing on the HP 735 (`dspsvr`), and displaying X and Starbase output on the X Station (`myxterm:0.0`).

Note The application in this example did not need to be re-linked, nor were any code changes necessary. This assumes that the application is linked with shared libraries, and the application uses `NULL` as the *driver* parameter to `gopen`.

How Does HP VMX Work?

Now that you have some understanding of what HP VMX is, and how you can use HP VMX, let us take a look at how HP VMX works.

Overview

Instead of rendering Starbase 3D graphics images to a dedicated graphics display subsystem, HP VMX is designed to display images in an X11 window using X11 protocol. It does this by automatically using one of the following methods, which are dependent on the attributes of the primitives it is rendering:

Method one	Simple graphics, such as lines and filled areas that do not require per-pixel computation (no depth buffering or shading, etc.), are rendered directly using X11 protocol. Note that this method will not be selected with dithering turned on. To ignore dithering or have it disabled, use the environment variable `HP_VM_XLIB_DITHER`. For information on the environment variable `HP_VM_XLIB_DITHER`, read the section "The Environment Variables `HP_VM_RENDER_METHOD` and `HP_VM_XLIB_DITHER`" in this chapter.
Method two	Primitives that require per-pixel computation are rendered into a virtual memory frame buffer and then copied to the window as a complete image with X11 protocol.

If the X11 window is local to a workstation, as it is when the HP VMX driver is rendering to a window in the overlay planes of the workstation, the HP VMX driver will always render into virtual memory and copy the finished image.

Here are the basic steps HP VMX performs to render into a virtual memory frame buffer and copy the image to a window:

1. **VM (Virtual Memory) frame buffer allocation** — At `gopen` time, the HP VMX driver allocates virtual memory for use as a frame buffer. This VM

frame buffer is allocated using `calloc` and its size is based upon the size of the X11 window being `gopen`ed.

2. **Rendering to the VM frame buffer** — After a successful `gopen`, HP VMX renders Starbase output primitives in the allocated VM frame buffer. The appropriate primitive attributes and device control are applied during rendering.
3. **Display of the VM frame buffer** — Upon application request, HP VMX displays the contents of the VM frame buffer. Application requests come in the form of one of the following Starbase calls:
   - `make_picture_current`
   - `flush_buffer`
   - `dbuffer_switch`

   See the section "Synchronization" for more details.

   Because HP VMX is always operating in the X11 windows environment, the display of the VM frame buffer to the `gopen`ed window is handled through the use of standard X11 protocol.

   These basic HP VMX steps are applicable to both single- and double-buffering.

HP VMX Configurations

The HP VMX driver supports depth 8 or depth 24 X11 windows. Attempts to `gopen` windows with a depth other than 8 or 24 will result in a Starbase error.

The HP VMX driver supports the following configurations:

- 8-bit indexed color (`CMAP_NORMAL`, `CMAP_MONOTONIC`), single-buffered, or 8/8 double-buffered
- 8-bit direct color (`CMAP_FULL`), single-buffered or 8/8 double-buffered
- 24-bit direct color (`CMAP_FULL`), single-buffered or 24/24 double-buffered

HP VMX Device Driver, VM Rendering Utilities, and Overlay Planes

Before going further, you need to understand the two basic functions that HP VMX provides:

- Rendering of Starbase graphics into a virtual memory frame buffer.
- Displaying of this virtual memory (VM) frame buffer in the targeted X11 window.

Together, these two functions create what we call "HP VMX".

VM Rendering Utilities

There exists, as a matter of implementation, a set of internal graphics system functions which rely on VM rendering, but not on the display of the VM buffer in a window. This set of functions is called the VM Rendering Utilities and includes:

VM Backing Store	Retain graphics data rendered to obscured portions of a window.
VM Double-Buffering	Allow low end systems to take advantage of double-buffering.

Again, these utilities are not included in the definition of HP VMX but do rely on some of the same internal implementation.

The majority of this chapter will discuss HP VMX as a "device driver" and is organized in a manner similar to the other device driver chapters.

The "VM Rendering Utilities" section near the end of the chapter discusses in more detail each of the VM rendering utilities and explains some of the implementation details.

Overlay Planes

HP VMX serves as the Starbase driver for all CRX-family and HCRX-family "overlay plane" device opens. Note, that the "hardware device driver" (for example, `hpgcrx` or `hpcrx48z`) is *not* supported in the overlay planes on these devices. HP VMX is used as the exclusive Starbase driver for the overlay planes on these devices.

Please see the section "HP VMX: The Overlay Plane Driver" for details on how HP VMX is used in this capacity.

HP VMX Performance

HP VMX performance is quite good. While HP VMX is generally slower than a hardware device driver, it provides 3D client/server graphics at an interactive performance level.

Rather than attempt a full performance characterization of HP VMX, this section contains some qualitative guidelines to use when assessing the performance of HP VMX. Performance on HP VMX as a whole is determined by the performance of the two key portions of HP VMX:

- **VM (Virtual Memory) rendering** — High performance VM rendering is achieved by taking advantage of HP's high performance SPUs and PowerShade graphics software.
- **Display of the VM frame buffer** — Display performance is difficult to characterize because it is influenced by the performance of the X11 servers and the network throughput.

  VM rendering and display performance are both influenced by the size of the graphics window. The larger the window, the more data there is to write to the VM frame buffer, and the more data there is to display via X11 protocol. HP VMX is optimized to display only the portions of the VM frame buffer that have changed since the last display.

X Windows

The HP VMX driver is supported only in the X11 window environment.

Compiling Your Application with the HP VMX Graphics Driver

You can find information for compiling your Starbase application with HP VMX shared and archived libraries in the *Graphics Administration Guide*.

To Open and Initialize the Device for Output

X11 Environment

The VMX usage example in the section "How Do You Use HP VMX?" gives an example of how you might run an application with HP VMX in the X11 environment. This section describes in more detail the X11 environment setup necessary to run HP VMX remotely.

DISPLAY Environment Variable

The `DISPLAY` environment variable must be set on the HP VMX server side. The value of the environment variable is the host, display, and screen of the targeted VMX client on which the Starbase application is to be displayed. By setting this environment variable, the application will direct X11 protocol to the HP VMX client.

For more information on setting the `DISPLAY` environment variable, read the section "Setting the DISPLAY Variable" in the chapter "Preliminary Configuration" of the *Using the X Window System* manual.

xhost Client

The `xhost` client is used to add or delete a remote host's permission to access the local display server. This client must be run on the HP VMX client side to allow the HP VMX server access to the HP VMX client's display server.

For more information on adding and deleting hosts with `xhost`, read the section "Adding and Deleting Hosts with `xhost`" in the chapter "Using the X Clients" in the *Using the X Window System* manual.

Syntax Examples

Two methods exist to `gopen` a window using HP VMX. The recommended method is to set the *driver* parameter to `NULL` and let Starbase choose the appropriate device driver. The second method for `gopen`ing HP VMX is to specify `hpvmx` as the *driver* parameter to `gopen()`. See the `gopen(3G)` and `inquire_device_driver(3g)` man pages for details on device driver selection.

If you specify `NULL` as the *driver* parameter, Starbase will choose HP VMX if:

- The window is displayed on a remote X11 server, or
- The window is displayed in the overlay planes on one of the CRX, HCRX, or HP VISUALIZE family of devices (for example, CRX-24Z, CRX-48Z, HCRX-8, HP VISUALIZE-24, and HCRX-24Z)

In each of the examples below, assume that the depth 8 or 24 window[3]:

⟨*screen*⟩`/remote_window`

has been created on a remote X11 server with the following `xwcreate` command:

`xwcreate -display` ⟨*remote_host*⟩ `-geometry 500×500 remote_window`

[3] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

Syntax Examples

The following examples show how to open the grapics devices for output[4]:

C programs

```
fildes=gopen("⟨screen⟩/remote_window", OUTDEV, NULL, INIT | THREE_D);
```

FORTRAN 77 programs

```
    fildes = gopen('⟨screen⟩/remote_window'//char(0), OUTDEV,
   +              char(0), INIT | THREE_D)
```

Pascal programs

```
fildes:=gopen('⟨screen⟩/remote_window', OUTDEV, '', INIT | THREE_D);
```

5

Parameters for gopen

The gopen procedure has four parameters: *path*, *kind*, *driver*, and *mode*.

- *path* — This is the name of the device file created by `xwcreate`(1) or created with `XCreateWindow`(3X11) and returned from:

  ```
  make_X11_gopen_string(3G)
  ```

- *kind* — This parameter should be `OUTDEV` if the window will be used for output, `INDEV` if the window will be be used for Starbase input, or `OUTINDEV` if the window will be used for both output and Starbase input.
- *driver* — The character representation of the driver type. For portability across the HP graphics device family, use the `NULL` parameter. In this case, Starbase will automatically choose the appropriate driver.

  For example,

`NULL`	*for C*
`char(0)`	*for FORTRAN 77*
`''`	*for Pascal*

[4] The actual path names of directories in angle brackets depend on the file system structure. See the *Graphics Administration Guide* for details.

A character string may be used to specify the driver. For example,

`"hpvmx"`	*for C*
`'hpvmx'//(0)`	*for FORTRAN 77*
`'hpvmx'`	*for Pascal*

- *mode* — The mode control word consists of several flags bits *OR*ed together. Listed below are flag bits that have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure. See the *Starbase Graphics Techniques* manual for more details of `gopen` actions when in an X Window.
  - `0` (zero) — Open the window, but do *not* perform the operations associated with `INIT` below. The following actions are:
    1. The software color table is initialized from the X color map already associated with the window.
    2. The VM buffer is initialized by reading the contents of the window.
  - `INIT` — Open and initialize as follows:
    1. The window is cleared to zeros.
    2. A new X color map is created and associated with this window. The color map is initialized as `CMAP_NORMAL`.
  - `RESET_DEVICE` — This flag is equivalent to `INIT`.
  - `SPOOLED` — Not supported.
  - `MODEL_XFORM` — Opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed. See `gopen(3G)` for more information.
  - `INT_XFORM` — Only integer and common operations will be performed. All floating point operations will cause an error.
  - `INT_XFORM_32` — Only integer and common operations will be performed. All floating point operations will cause an error.
  - `ACCELERATED` — This flag is ignored.
  - `UNACCELERATED` — This flag is ignored.
  - `THREE_D` — Three-dimensional graphics.

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location (0, 0) is the upper-left corner of the window with X-axis values increasing to the right and Y-axis values increasing down.

Use this form of pixel addressing when calling high-level Starbase operations in terms of (`x,y`) device coordinates.

The maximum `x` and `y` coordinates are determined by the size of the window.

Device Defaults

Raster Echo Default

The default raster echo is the following 8x8 array:

255	255	255	255	0	0	0	0
255	255	0	0	0	0	0	0
255	0	255	0	0	0	0	0
255	0	0	255	0	0	0	0
0	0	0	0	255	0	0	0
0	0	0	0	0	255	0	0
0	0	0	0	0	0	255	0
0	0	0	0	0	0	0	255

The maximum size for a raster echo is 64x64 pixels.

Semaphore Default

Semaphore operations have no effect on HP VMX.

Line Type Defaults

The default line types are created with the bit patterns shown below:

Table 5-1. Line Type Defaults

Line Type	Pattern
0	1111111111111111
1	1111111100000000
2	1010101010101010
3	1111111111111010
4	1111111111101010
5	1111111111100000
6	1111111111110100
7	1111111110100110

Color

Default Color Map

To initialize the current color map to the default values shown below, set the *mode* parameter of `gopen` to `INIT` when opening a depth 8 window. This is the Starbase `CMAP_NORMAL` mode. (To see the rest of the colors, use the `inquire_color_map` call to read the Starbase color table).

Table 5-2. Starbase Default Color Table

Index	Color	Red	Green	Blue
0	black	0.0	0.0	0.0
1	white	1.0	1.0	1.0
2	red	1.0	0.0	0.0
3	yellow	1.0	1.0	0.0
4	green	0.0	1.0	0.0
5	cyan	0.0	1.0	1.0
6	blue	0.0	0.0	1.0
7	magenta	1.0	0.0	1.0
8	10% gray	0.1	0.1	0.1
9	20% gray	0.2	0.2	0.2
10	30% gray	0.3	0.3	0.3
11	40% gray	0.4	0.4	0.4
12	50% gray	0.5	0.5	0.5
13	60% gray	0.6	0.6	0.6
14	70% gray	0.7	0.7	0.7
15	80% gray	0.8	0.8	0.8
16	90% gray	0.9	0.9	0.9
17	white	1.0	1.0	1.0
⋮	⋮	⋮	⋮	⋮
255	white	1.0	1.0	1.0

Starbase Functionality

Calls Not Supported

The `hpvmx` driver does not support the following Starbase calls if you are using Starbase without the PowerShade software. When executed, these calls will produce no result (that is, they are ignored).

`alpha_transparency`	`hidden_surface`
`backface_control`	`light_ambient`
`bf_alpha_transparency`	`light_attenuation`
`bf_control`	`light_model`
`bf_fill_color`	`light_switch`
`bf_interior_style`	`line_filter`
`bf_perimeter_color`	`perimeter_filter`
`bf_perimeter_repeat_length`	`set_capping_planes`
`bf_perimeter_type`	`set_model_clip_indicator`
`bf_surface_coefficients`	`set_model_clip_volume`
`bf_surface_model`	`surface_coefficients`
`bf_texture_index`	`surface_model`
`contour_enable`	`texture_index`
`define_contour_table`	`texture_viewport`
`define_texture`	`texture_window`
`define_trimming_curve`	`viewpoint`
`deformation_mode`	`zbuffer_switch`

Using PowerShade with HP VMX

By adding PowerShade's capabilities to HP VMX, a much wider set of functionality is supported. The following calls are not supported when using PowerShade with HP VMX:

`alpha_transparency`	`line_filter`
`bf_alpha_transparency`	`perimeter_filter`
`bf_texture_index`	`texture_index`
`contour_enable`	`texture_viewport`
`define_contour_table`	`texture_window`
`define_texture`	

Conditional Support of Starbase Calls with HP VMX

The following Starbase calls are supported with the listed exceptions:

`block_read`, `block_write`

The *raw* parameter for the `block_read` and `block_write` commands is used by this driver to do plane-major reads and writes. It is enabled by the gescape `R_BIT_MODE`.

The storage destination supplied by the user as the source or destination must be organized as follows.

- The data from each plane is packed with eight pixels per byte.
- Each row must begin on a byte boundary. Thus the size of the rectangle as specified by the ⟨*length_x*⟩ and ⟨*length_y*⟩ parameters must correspond to an integral number of bytes.
- The data for the next plane begins on the following byte boundary.
- Clip to the screen limits.
- The first pixel in the source rectangle is placed in the high-order bit of the first byte in each plane region.
- When clipping, part of each plane region will not be read (`block_read`) or altered (`block_write`).

A bit mask selects the planes to read or write. The initial value of this mask is 1 (one) indicating that only plane 0 is to be accessed. The value of the mask may be changed using the `R_BIT_MASK` or `GR2D_PLANE_MASK` `gescape`s. `GR2D_PLANE_MASK` is discussed in Appendix A of this manual. The planes selected by the mask are expected to reside in consecutive plane locations in the user storage area. This reduces the storage requirements to exactly what is needed but also presents the potential for address violations or undesirable results.

For example, if the plane mask is changed to specify more planes between a `block_read` and a following `block_write` from the same location, the `block_write` will attempt to

	access storage for planes that were not read (and perhaps not allocated). The application program must ensure consistency in these operations.
`double_buffer`	HP VMX supports only 8/8 double-buffering in a depth 8 window, and only 24/24 double-buffering in a depth 24 window.
`fill_dither`	The ability to dither is disabled if the number of colors specified by `fill_dither` is one. However, if the number of colors specified is greater than 1, the default dither cell size of 16 is used. Dithering is only used in depth 8 windows while in either the `CMAP_FULL` or `CMAP_MONOTONIC` color map mode.
`pattern_define`	For HP VMX, the maximum pattern size is 4x4. If a pattern larger than 4x4 is specified, an error message is printed and the previous pattern is retained.
`shade_mode`	The color map mode may be selected. Shading can be turned on only if using PowerShade. Shading is not supported on device coordinate primitives even with PowerShade.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	If *not* using PowerShade software, `vertex_format`'s *use* parameter must be set to zero. Any extra coordinates will be ignored. If using PowerShade software, `vertex_format` is fully functional.

`*_with_data`

The following routines are called `with_data` routines because they contain `with_data` in their routine names. These `with_data` routines are supported by HP VMX:

```
partial_polygon_with_data3d
polygon_with_data3d
polyhedron_with_data
polyline_with_data3d
polymarker_with_data3d
polyquad_with_data3d
polytriangle_with_data3d
quadrilateral_mesh_with_data
triangle_strip_with_data
```

Note that the `TEXTURE_MAP` flag applies to the TurboVRX devices via the `texture_*` routines. This extra data per vertex is not used in the `tm_*` routines.

For detailed information on these routines, read the *Starbase Reference* and "Appendix A" of this document. In some cases, you will be able to find the routines under their own name, but in other cases, you will need to use the first part of the routine name to locate these routines (e.g., `polyline_with_data3d` is described on the man page for `polyline(3G)`).

Supported Gescapes

The `hpvmx` driver supports the following gescape operations. Refer to Appendix A found in the *HP-UX Starbase Device Drivers Manual* for details on gescapes.

`BAD_SAMPLE_ON_DIFF_SCREEN`	Restore the locator and choice sampling of the X11 pointer device.
`BLOCK_WRITE_SKIP_COUNT`	Specify byte skip count during block write.
`DRAW_POINTS`	Select different modes of rounding for rendered points.
`IGNORE_RELEASE`	Trigger only when button is pressed.
`OLD_SAMPLE_ON_DIFF_SCREEN`	Inquire the locator and choice sampling of the X11 pointer device.
`R_BIT_MASK`	Define bit mask for bit mode block operations.
`R_BIT_MODE`	Enable/disable bit mode block operations.
`R_GET_FRAME_BUFFER`	Read the address of the device frame buffer and control space.
`R_LINE_TYPE`	User-defined line style and repeat length.
`READ_COLOR_MAP`	Copy the hardware color map into the software color map.
`TRIGGER_ON RELEASE`	Trigger only when button is released.

Additional Gescapes Supported if PowerShade is Enabled

`ILLUMINATION_ENABLE`	Turn on/off illumination bits.
`LS_OVERFLOW_CONTROL`	Set light source overflow handling.
`POLYGON_TRANSPARENCY`	Segment control over front/back face "screen door."
`TRANSPARENCY`	Set screen door transparency mask (front face and back face).
`WIDELINE_CONTROL`	Turn on/off and set attributes of widelines.
`ZBANK_ACCESS`	Enable/disable Z-buffer block operations.
`ZWRITE_ENABLE`	Enable/disable replacement of Z value.

Exceptions to Gescape Support

Note Because the gescape operations are device-dependent, the exceptions discussed below may be removed in future drivers.

`R_GET_FRAME_BUFFER`	This gescape is used to return the addresses of both the frame buffer and the control space. A zero is returned for the control space address since HP VMX has no control space. The frame buffer address is returned correctly.
`R_LOCK_DEVICE`, `R_UNLOCK_DEVICE`, `SWITCH_SEMAPHORE`	Because HP VMX renders to a virtual memory frame buffer, device locking is not necessary. Therefore, these gescapes have no effect on HP VMX.

Differences From Other Starbase Device Drivers

Synchronization

Because of the way HP VMX works (rendering to a VM buffer and then displaying these images through X11 protocol), the HP VMX driver has some unique synchronization requirements.

The following Starbase calls copy the contents of the VM frame buffer to the window:

- `dbuffer_switch` (if double-buffering is enabled)
- `flush_buffer`
- `make_picture_current`

Until one of these calls is made, HP VMX will continue to render graphics to the VM frame buffer. Changes are not reflected in the X11 window until this synchronization occurs.

You may use `buffer_mode(3G)` to disable buffering of graphics primitives, and therefore avoid synchronization problems. Disabling buffering with `buffer_mode` will degrade performance.

Note that there is no command buffer associated with the HP VMX driver. When `buffer_mode` is turned off, there is an implicit `make_picture_current` which causes an update of the virtual memory buffer to the destination window. It is the frequency of these updates (that is, synchronization) that can degrade rendering performance significantly.

The *Starbase Reference* and the *Starbase Graphics Techniques* manuals discuss `buffer_mode` for bit-map device drivers.

Resource Considerations

Some resource usage implications need to be considered when using the HP VMX driver. Because no dedicated frame buffer hardware exists, and therefore the frame buffer memory is allocated at `gopen` time, the use of the HP VMX driver will consume virtual memory resource.

HP VMX will allocate a virtual memory frame buffer at `gopen` time. The VM frame buffer is allocated based upon the size of the X11 window being `gopen`ed.

Since HP VMX supports only depth 8 or depth 24 X11 windows, the frame buffer is allocated on a byte-per-pixel or word-per-pixel basis.

For example, consider a depth 8 window which is 750 pixels wide and 600 pixels high. The size of the VM frame buffer is:

750 pixels $\times$ 600 pixels = 450,000 pixels

450,000 pixels $\times$ 1 byte/pixel = 450,000 bytes

So the VM frame buffer for this window consumes approximately .45 Mbytes of virtual memory.

A depth 24 window uses 32 bits/pixel, so will consume approximately four times as much virtual memory as an 8-bit window of the same size.

This resource is returned to the system at `gclose()` time.

This resource usage is typically not a problem, but should be considered if you are using the HP VMX driver and one of the following conditions exists:

- Several windows (especially large windows) are `gopen`'d simultaneously.
- Your system has a small amount of physical memory.

These resource conditions could cause:

- System errors
- Application termination
- Performance degradation.

In order to alleviate these resource conditions, you should:

- Use the hardware device driver whenever possible. For example, if your display is an HCRX graphics device, specify `hphcrx` as the *driver* parameter to `gopen` when running to a local window. Specify `hpvmx` only when running to a remote system, or when running to the overlay planes. (If your *driver* parameter is `NULL`, this happens automatically.)
- Reduce the number of simultaneous `gopens` (`gclose` some windows before `gopen`ing more of them).

- Use smaller windows (the size of the window determines the amount of memory allocated at `gopen` time).
- Add more memory to your system.
- Increase the size of the kernel's maximum data size parameter (`maxdsiz`).

Restricted gopens

As with Virtual Memory (VM) double-buffering, multiple `gopen`s of HP VMX to the same window by different processes should not be attempted. The VM frame buffer allocated by HP VMX is associated with each process rather than with a window. Therefore, multiple `gopen`s to the same window will each allocate a new VM buffer, rather than "share" one VM buffer. This will produce results potentially different from other hardware devices or expectations.

VM Rendering Utilities

Note that this section covers VM Rendering Utilities, *not* the HP VMX driver functionality. Also note that the VM Rendering Utilities are only used with depth 24 windows.

As mentioned in the section "HP VMX Device Driver, VM Rendering Utilities, and Overlay Planes" Starbase implements a set of VM Rendering Utilities which rely on a portion of the HP VMX functionality.

Recall that HP VMX performs two basic functions:

- Renders Starbase graphics into a virtual memory frame buffer.
- Displays this VM frame buffer in the targeted X11 window.

The set of VM rendering utilities exercises only the first of these two HP VMX functions – the rendering of Starbase graphics into a virtual memory frame buffer. The method of display is *not* handled by HP VMX, but by the methods described in the subsequent sections. This section takes a look at the VM rendering utilities and briefly explains their implementation.

Note, while these actions are largely internal implementation details, they are worth discussion here so that you recognize the similarities between their implementation and the use of HP VMX as a device driver.

VM Double-Buffering on 8-plane devices

Virtual Memory Double-Buffering

4/4 double-buffering limitations

Where double-buffering on Series 700 models with integrated and internal color graphics is possible, it is limited. As 8-plane devices, these models only allow 4/4 double-buffering. You are limited to 16 colors as rendering in this mode uses four planes per buffer. Also, X11 does not support 4/4 double-buffering, so where your graphics window double-buffers as expected, the rest of your windows may flash (known as the technicolor effect). Note that 4/4 double-buffering is not supported with `CMAP_FULL` mode.

8/8 double-buffering enhanced performance

Virtual memory (VM) 8/8 double-buffering is supported by setting the `HP_VM_DOUBLE_BUFFER` environment variable to `TRUE`. This functionality allows you to double-buffer in 8 planes per buffer, giving you access to 256 colors. It is also supported by X11 so technicolor is not a problem.

Here's how it works:

The virtual memory buffer is allocated by the Starbase graphics library to mirror the window. The VM rendering capabilities of HP VMX are used to render the Starbase graphics images into the allocated virtual memory buffer. The Starbase graphics library then copies the VM buffer (containing the Starbase graphics output) to the display frame buffer at `dbuffer_switch` time.

Be aware of tradeoffs

VM double-buffering is not appropriate for all applications. You should first evaluate the performance of your application against the following tradeoffs:

1. Speed – VM rendering uses only software rendering. As a result, rendering to the VM buffer is somewhat slower for many operations and significantly slower for a few operations such as drawing non-Z-buffered, non-shaded vectors.
2. More memory – A VM double-buffering application uses more virtual memory in order to allocate the VM buffer. The size of this buffer is proportional to the size of the window when it was `gopen`ed for rendering. The buffer size is one byte for each pixel in the window. If the `SUPPRESS_CLEAR` double-buffering flag is set, then the VM buffer will be double the size of the window. Note

that the buffer memory is returned to the system when the application process terminates; it does not stay allocated with the window. (Most applications do not need to change the kernel configuration to use this capability. If your application has problems, you can increase the kernel's `maxdsiz` parameter using `SAM`(1)).

3. `SUPPRESS_CLEAR` – As of HP-UX 9.05, Starbase VM (virtual memory) double-buffering supports `SUPPRESS_CLEAR`. If the *mode* parameter of the `double_buffer` command is ORed with the `SUPPRESS_CLEAR` flag, then the buffer that is enabled for writing will not be cleared by subsequent calls to the `dbuffer_switch` command. Also, multiple Starbase `gopen` calls in the same process to the same VM double-buffered window will use the same VM double-buffer (rather than a new VM double-buffer for each `gopen` call).

To enable VM double-buffering

There are two ways you can enable VM double-buffering on Internal Color and Integrated Graphics Workstations (only true for these workstations).

- You need to define the `HP_VM_DOUBLE_BUFFER` variable in your environment *before* starting your application. For example, using `ksh` syntax, execute the following:

  ```
  export HP_VM_DOUBLE_BUFFER=TRUE
  ```

- The application can define the environment variable itself before `gopen`ing the window using the `putenv(3c)` function.

Once VM double-buffering is enabled as above, the Starbase `double_buffer` function accepts 8 planes to be specified in the *planes* parameter. If VM double-buffering is not enabled, the `double_buffer` function limits you to 4 planes.

VM Backing Store

The *Starbase Graphics Techniques* manual gives a good explanation of backing store (also known as "retained raster"). Backing store is memory used to retain graphics data rendered to obscured portions of a window. This memory is allocated by the X server. The VM rendering capabilities of HP VMX are used to render the Starbase graphics images into the allocated backing store memory. The X server is then responsible for copying this backing store memory to the window when the obscured regions are exposed.

Refer to the "Backing Store Operation" section of the *Starbase Graphics Techniques* manual for more information on backing store.

HP VMX: The Overlay Plane Driver

As mentioned in the section "HP VMX Device Driver, VM Rendering Utilities, and Overlay Planes," HP VMX serves as the Starbase driver for all CRX-family and HCRX-family *overlay plane* device opens. The "hardware device driver" for these devices (e.g. `hpgcrx` or `hpcrx48z`) is *not* supported in the overlay planes. HP VMX is used as the exclusive Starbase driver for the overlay planes on these devices.

If you `gopen` a window in the overlay planes of a CRX-family or HCRX-family device with a `NULL` *driver* parameter, the `hpvmx` driver will be selected. Alternatively, you may explicitly ask for the HP VMX driver by specifying `hpvmx` for the *driver* parameter to `gopen`.

Note that 8/8 double-buffering is supported in the overlay planes using HP VMX.

Table 5-3 details driver selection for the CRX-family of devices at `gopen` time. Note, the driver selected is based on whether:

- A window is in the image planes or overlay planes.
- The `hphcrx`, `hpgcrx`, `hpcrx48z`, `hpvmx`, or `NULL` driver is specified.

Table 5-3. Driver Selection at gopen

Type of Window	**Driver Specified**	**Window Depth**	**Driver Used**
Overlay	`NULL`	8	`hpvmx`
Overlay	`hphcrx`, `hphcrx48z`, `hpgcrx`, `hpcrx48z`, or `hpvisx`	8	not supported
Overlay	`hpvmx`	8	`hpvmx`
Image	`NULL`	8, 12, 24	`hphcrx`, `hphcrx48z`, `hpgcrx`, or `hpcrx48z`
Image	`hphcrx`, `hphcrx48z`, `hpgcrx`, or `hpcrx48z`	8, 12, 24	`hphcrx`, `hphcrx48z`, `hpgcrx`, `hpcrx48z`, or `hpvisx`
Image	`hpvmx`	8, 24	`hpvmx`
		12	not supported

SOX11 vs. HP VMX

As mentioned earlier, HP VMX and the Starbase on X11 (SOX11) drivers are similar in that both provide Starbase functionality in the X11 client/server model. This section will briefly compare and contrast the two drivers.

Functionality

The most significant difference between HP VMX and SOX11 is that PowerShade is supported on HP VMX, but is *not* supported on SOX11. This results in a much richer set of Starbase functionality for HP VMX, including lighting, shading, and hidden surface removal. These features are *not* supported on SOX11.

Performance

When comparing the functionality of the HP VMX driver to the SOX11 driver, the HP VMX driver provides better performance than the SOX11 driver. However, the HP VMX driver can be slower than the SOX11 driver if the image being displayed is sparse and conditions force HP VMX to use virtual memory and copy rendering.

The HP VMX driver always allocates a virtual memory frame buffer. The SOX11 driver does not use a virtual memory frame buffer, which can save considerable memory, especially if an application opens multiple windows.

Changing from HP VMX to SOX11

If you prefer to use the `sox11` driver instead of the HP VMX driver, there are two ways to do this:

- Set the `SB_DEFAULT_SOX11` environment variable to any non-null value. This only applies when the *driver* parameter of `gopen` is `NULL`.
- Set the *driver* parameter of your `gopen` statement to `sox11`.

For more information on the HP VMX and SOX11 drivers, read the chapter "The Starbase-on-X11 Device Driver" found in the *HP-UX Starbase Device Drivers Manual*.

The Environment Variables HP_VM_RENDER_METHOD and HP_VM_XLIB_DITHER

The environment variables discussed in this section help you take advantage of the new HP VMX behavior discussed in the section “How Does HP VMX Work” in this chapter.

HP_VM_RENDER_METHOD

In previous releases, the HP VMX driver always rendered into the virtual memory frame buffer for all primitives. However, in 10.01 and later releases of HP-UX, the HP VMX driver can render simple primitives directly with X11 protocol. If you want to disable this new behavior to guarantee pixel-for-pixel compatibility, set the environment variable `HP_VM_RENDER_METHOD` to `PIXMAP`.

HP_VM_XLIB_DITHER

The HP VMX driver selects virtual memory rendering instead of X11 protocol rendering if dithering is enabled. In Starbase, this is controlled with the `fill_dither` call. If you need to ignore dithering in order to allow X11 protocol rendering for better performance of simple wireframe drawings, set the environment variable `HP_VM_XLIB_DITHER` to `IGNORE` before starting the application. Note, however, if you ignore dithering, your color approximation capabilities will be less accurate until you `unset` this variable. This environment variable only affects primitives that are rendered with X11 protocol.

6

Gescapes

Introduction

This chapter contains information regarding new Starbase gescapes for the HP-UX 10.20 and the July, 1997 Workstation ACE for 10.20 HP-UX releases.

R_READ_FB

Supported Devices

To determine if this gescape is supported on your device, use the routine `inquire_capabilities`(3g) to check that the `COLOR_2_CAPABILITIES` flag has the `IC_RGB_IO` bit set to `1` (one).

Description

The `R_READ_FB` gescape allows you to read the image out of a window created with the shade mode set to `CMAP_FULL`. The range to read is specified in device coordinates similar to `dcblock_read`(3g). The result will be stored in 24-bit RGB-per-pixel in the order of right to left and top to bottom, regardless of the current visual. The resulting 24-bit image will appear as close as possible to the contents of the frame buffer. For example, an image read from an 12-bit full color visual will be expanded to 24 bits of information. Since the coordinates relate to the device and not the drawing area, the current clip rectangle will not affect the results.

Application developers using this gescape should keep the following in mind:

- The results will be the same regardless of the current clipping level or clipping rectangle.
- The current replacement rule as set by `drawing_mode`(3g) will affect the results in the same manner as `dcblock_read`(3g). If you want the exact contents of the frame buffer, the replacement rule should be set to the default (`SOURCE`).
- Do not set the `write_enable`(3g) and `display_enable`(3g) masks when using this gescape.
- If HP Color Recovery is enabled and you are reading from a depth 8 visual, then operation of this gescape will be slow because of the use of a complex digital filter to convert 8-bit data stored in the frame buffer into 24-bit per pixel color data in memory.
- If you specify an area to read which is larger than the window, only the part of your buffer which intersects the window will be changed. In other words, the entire area will not be changed in this case; just the part of your specified area that intersects the window.
- If a second window obscures the window that you are reading, the results may be clipped. An area that is clipped in this manner will be black. To avoid this artifact, bring the window to the front before reading.

The data parameters passed to the `gescape` entry point are used as follows. Note that both *arg1* and *arg2* are used to pass information into the `gescape`() entry point.

`arg1.i[0]`	The starting X location, in pixels.
`arg1.i[1]`	The starting Y location, in pixels.
`arg1.i[2]`	The height of the area to be read, in pixels.
`arg1.i[3]`	The width of the area to be read, in pixels.
`arg1.i[4]`	The array into which the image should be read.

C Syntax Example

```
char flags[SIZE_OF_CAPABILITIES];
gescape_arg arg1;
unsigned char image[512*512*3];

inquire_capabilities (fildes, SIZE_OF_CAPABILITIES, flags);
if (flags[COLOR_2_CAPABILITIES] & IC_RGB_IO) {
  arg1.i[0] = 0;      /* starting x */
  arg1.i[1] = 0;      /* starting y */
  arg1.i[2] = 512;    /* width */
  arg1.i[3] = 512;    /* height */
  arg1.i[4] = (int)image;     /* RGB results */
  gescape (fildes, R_READ_FB, &arg1, NULL);
}
```

R_WRITE_FB

Supported Devices

To determine if this gescape is supported on your device, use the routine `inquire_capabilities`(3g) to check that the `COLOR_2_CAPABILITIES` flag has the `IC_RGB_IO` bit set to `1` (one).

Description

The `R_WRITE_FB` gescape allows you to write an image to a window with shading mode set to `CMAP_FULL`. The range to write is specified in device coordinates similar to the function `dcblock_write`(3g). The data to write is contained in a byte array with the first three bytes containing the 8 bit red, green and blue values respectively. Pixels (to write) should be stored in the following order: left to right along a row, with top to bottom ordering of rows. The first three bytes of data in the array correspond to the RGB data for the pixel in the upper left corner of the destination block. This gescape will convert the 24-bit image into a form compatible with the current visual and write out the results. The image buffer will not be changed. The resulting 24-bit image will appear as close as possible to the contents of the image buffer. Since the coordinates relate to the device and not the drawing area, the current clip rectangle will not affect the results.

Application developers using this gescape should keep the following in mind:

- The results will be the same regardless of the current clipping level or clipping rectangle.
- The current replacement rule as set by `drawing_mode`(3g) will affect the results in the same manner as `dcblock_write`(3g). If you want your image to be displayed without this modification, the replacement rule should be set to the default (`SOURCE`).
- Do not set the `write_enable`(3g) and `display_enable`(3g) masks when using this gescape.
- Using this gescape in a visual with less than 24 bits will cause your image to be dithered. Although this provides good performance, the resulting image cannot be exactly as specified. If you are using a device with Color Recovery turned on, the image to write is dithered and then filtered. The image can be written quickly and will look similar to a 24-bit image. However, if you read the image and then write the image you just read, the filtering algorithm in the hardware will have been performed on the image twice and may degrade the image.
- If you specify an area to read which is larger than the window, only the part of your buffer which intersects the window will be changed. In other words, the entire area will not be changed in this case; just the part of your specified area that intersects the window.
- If a second window obscures the window that you are writing, the results may be clipped. To avoid this problem, bring the window to the front before reading.

The data parameters passed to the `gescape` entry point are used as follows. Note that both *arg1* and *arg2* are used to pass information into the `gescape`() entry point.

`arg1.i[0]`	The starting X location, in pixels.
`arg1.i[1]`	The starting Y location, in pixels.
`arg1.i[2]`	The height of the area to be read, in pixels.

`arg1.i[3]`	The width of the area to be read, in pixels.
`arg1.i[4]`	The array from which the image should be read.

C Syntax Example

```
char flags[SIZE_OF_CAPABILITIES];
gescape_arg arg1;
unsigned char image[512*512*3];

inquire_capabilities (fildes, SIZE_OF_CAPABILITIES, flags);
if (flags[COLOR_2_CAPABILITIES] & IC_RGB_IO) {
  arg1.i[0] = 0;      /* starting x */
  arg1.i[1] = 0;      /* starting y */
  arg1.i[2] = 512;    /* width */
  arg1.i[3] = 512;    /* height */
  arg1.i[4] = (int)image;     /* Image to write */
  gescape (fildes, R_WRITE_FB, &arg1, NULL);
}
```

SET_POLYGON_OFFSET

Supported Devices

As of this printing, the `SET_POLYGON_OFFSET gescape` is supported on the following graphics devices. Since this list may change, you should use `inquire_capabilities` to determine support for your device, as shown in the example that follows the description of this gescape.

- HCRX-8Z
- HCRX-24Z
- CRX-48Z
- HP VISUALIZE-8
- HP VISUALIZE-24
- HP VISUALIZE-48
- HP VISUALIZE-48XP
- HP VISUALIZE-FX^2
- HP VISUALIZE-FX^4
- HP VISUALIZE-FX^6

Description

The ⟨*op*⟩ parameter is `SET_POLYGON_OFFSET`.

`SET_POLYGON_OFFSET` causes the interior pixels of front- and back-facing area primitives (for example, polygons, triangular strips, quadrilateral meshes, polyhedra, and other such primitives), when in interior style `INT_SOLID` or `INT_TEXTURE`, to generate Z-buffer values that are offset (away from the viewer) from what the default Z-buffer values would normally be. This behavior allows an application to use an algorithm that may yield significantly better performance in rendering filled areas with edging (on those graphics devices that support the `gescape`) over the default Starbase method for rendering edged areas. Details of the algorithm are explained below. The `gescape` can also be used to eliminate some rendering artifacts. (An example is also described below.) This `gescape` is only useful when hidden surface rendering (HSR) is enabled (see `hidden_surface`(3g));

An application can call `inquire_capabilities`(3g) and check the `IC_POLYGON_OFFSET` bit in the `PRIMITIVES_2_CAPABILITIES` byte in order to determine if this `gescape` opcode is implemented on a particular `gopen` file descriptor.

Note that the offset is applied in the device coordinate (DC) Z-axis only, not in any geometric space such as modeling coordinates or world coordinates. Thus, it displaces the rendered pixels after all modeling, viewing, and viewport transformations have been applied.

The offset value is computed from two parts:

- A fixed bias that is always applied. The bias is specified as a device-independent floating point value. Starbase multiplies this value by the device-specific Z-buffer increment value. Thus, a bias value of `1.0` is typical.
- A factor that Starbase multiplies by each planar facet's maximum Z-gradient, with respect to the DC X or Y axes. Areas that are orthogonal to the viewing direction have a Z-gradient of zero, so the factor has no effect. Areas that slope sharply away from the viewpoint have large Z-gradients so the factor value adds a significant additional offset. The factor is specified as a floating point value without units; a starting value of `1.0` is suggested, but depending on the nature of the geometry and the viewing transformation, an adjustment to achieve the desired rendering effect may be required.

The results of the bias computation and the factor computation are summed to create the DC Z offset that is applied to the area primitive. Positive bias and factor values result in Z-buffer values that are "farther away" from the viewer; this is the normal usage. The results are undefined for non-planar facets, as a single Z-gradient cannot be computed for them. Offset areas are clipped to the device's DC Z-buffer limits.

This `gescape` also requires an integer flag value indicating whether polygon offset is to be enabled or disabled. The bias and factor values are only used when the enable flag is set; however, they should always be given valid floating point values.

The data parameters passed to the `gescape` entry point are used as follows. Note that both *arg1* and *arg2* are used to pass information into the `gescape`() entry point.

`arg1.i[0]`	The enable flag: set to `TRUE` to enable polygon offset; set to `FALSE` to disable application of the offset.
`arg2.f[0]`	The bias value.
`arg2.f[1]`	The factor value.

C Syntax Example

```
gescape_arg arg1, arg2;

arg1.i[0] = TRUE;
arg2.f[0] = 1.0;
arg2.f[1] = 1.0;
gescape (fildes, SET_POLYGON_OFFSET, &arg1, &arg2);
```

FORTRAN77 Syntax Example

```
integer*4 arg1(64)
real  arg2(64)

arg1(1) = 1
arg2(1) = 1.0
arg2(2) = 1.0
call gescape(fildes, SET_POLYGON_OFFSET, arg1, arg2)
```

Pascal Syntax Example

```
{gescape_arg is defined in starbase.p1.h}

var
   arg1, arg2: gescape_arg;
begin
   arg1.i[1] = 1;
   arg2.f[1] = 1.0;
   arg2.f[2] = 1.0;
   gescape (fildes, SET_POLYGON_OFFSET, arg1, arg2);
```

Improving Rendering of Edged Polygons

The following information describes how to use the `SET_POLYGON_OFFSET gescape` to improve the rendering of edged polygons.

The normal way to render edged areas in Starbase (and the way that must still be used in cases where the `SET_POLYGON_OFFSET` gescape is not supported) is to set the interior style to `INT_SOLID` and enable edging. Starbase renders the interior pixels in the fill color and the edge pixels in the edge color. Special rasterization is done to avoid "stitching" of edges. "Stitching" occurs when scattered pixels of a primitive (in this case an edge vector) are not drawn because the Z-buffer values at those pixels already indicate that the primitive is "obscured" (in this case by the interior fill pixels).

Better performance can be achieved in rendering edged areas by filling many areas, and then rendering all the edges as polyline primitives in a second pass. Improving performance also distributes the cost of modifying the graphics pipeline state from "fill" mode to "vector" mode over many primitives, rather than switching modes during each area primitive. Such a grouping of operations must be done at the application level. Note that the edging is still done via line primitives in this algorithm, so line attributes must be set to the desired edge values.

The typical problem with this better-performing method of rendering is that when the edge vectors are rendered, stitching may be visible because of the values already stored in the Z-buffer by the fill rendering. Offsetting the fill rendering in the Z-buffer can eliminate this stitching. Thus, this `gescape` opcode allows for better-looking images using a faster rendering method. Note that the per-area polygon offset computation does slightly slow down the rendering of filled areas, but for applications that can render significant numbers of area primitives followed by a few polyline primitives with many vertices (hundreds, perhaps), the grouping of area primitives and of polylines more than makes up for the computation overhead.

When using polygon offsets, artifacts in hidden-surface rendering can be introduced. For example, if a solid object such as a cube is being rendered, then depending on the angle of the view, one side might have a higher Z-gradient than an adjoining side. Because the more sharply-angled side could be offset more (depending on the offset factor), all the pixels of the adjoining side might not be obscured. This could result in ragged joints, especially with back-facing parts of the solid. One way to avoid this particular artifact, if the geometry to be rendered is appropriate, is to enable back-face culling, which is a recommended practice in any case for performance reasons. Artifacts can also occur in the intersection of filled areas with each other, or with other primitives.

As an example, here is a partial code skeleton that shows how an application can make use of polygon offset for the edging of filled areas. Note that this sample code is only intended to show the basic logic; it may not be the most efficient code design in terms of geometry management, or for avoiding unnecessary attribute changes.

```
int fildes;
unsigned char capabilities[SIZE_OF_CAPABILITIES];
int actual_size;
gescape_arg arg1, arg2;

.
.
.
fildes = gopen(...) /* Gopen the graphics device. */

/* Inquire whether polygon offset is supported on this gopen context. */
actual_size = inquire_capabilities(fildes, SIZE_OF_CAPABILITIES, capabilities);
.
.
.
/* Enable hidden surface rendering (i.e., use of the Z-buffer). Also */
/* enable back-face culling for performance, and to eliminate the most */
/* common polygon-offset artifact. */
hidden_surface (fildes, TRUE, TRUE);
.
.
.
/* Set up fill attributes (other than interior style). */
.
.
.
if (capabilities[PRIMITIVES_2_CAPABILITIES] & IC_POLYGON_OFFSET) {

   /* Since the gescape is supported, use the faster algorithm. */

   /* Set up line attributes with desired edge attribute values. */

   /* Set the interior style to solid, without edging. */
   interior_style (fildes, INT_SOLID, FALSE);

   /* Enable polygon offset. */
   arg1.i[0] = TRUE;
   arg2.f[0] = 1.0;
   arg2.f[2] = 1.0;
   gescape (fildes, SET_POLYGON_OFFSET, &arg1, &arg2);

   /* Collect area primitive geometry for edges to be rendered into */
   /* polyline geometry format.  (In some cases, the same geometry */
   /* arrays can be used for both filling and edging passes.)  */
   .
   .
   .
   /* Fill:  Render primitives with or without edge flags. (They will be */
   /* ignored.)  */

   ... /* Area primitive calls */

   /* Edge:  Render the edge geometry with or without move/draw flags. */

   ... /* Polyline calls */
}
else {
   /* Since the gescape is not supported, let Starbase draw the edges. */

   /* Set up edge attributes. */

   /* Set the interior style to solid, with edging. */
   interior_style (fildes, INT_SOLID, TRUE);

   /* Fill and Edge:  Render primitives with or without edge flags. */

   ... /* Area primitive calls */
}
```

Reducing Rendering Artifacts

The following information describes how to use this gescape to reduce rendering artifacts.

Stitching can occur during rendering in situations other than edging of filled areas. For example, perhaps surfaces are being rendered in `INT_SOLID` mode and it is desirable to put Starbase text labels or annotation on those surfaces. Drawing stroked text in the same geometric plane as the filled surface will often result in the text strokes being stitched. The following are common solutions for reducing stitching:

- Geometrically offset the plane in which the text is drawn slightly from the surface. For example, if the facets of a cube are being labeled, the text might be drawn slightly outside of each facet of the cube (in the direction of that facet's normal). This makes the text "float" over the facets of the cube.
- Use `SET_POLYGON_OFFSET` to eliminate the stitching in the text. This might cause labeling on areas that face away from the viewer to be visible, which may or may not be desirable behavior in a particular application. The advantage of this method is that no computations need to be done to offset the text geometry from the surface geometry.

Other similar examples may arise where this gescape may be useful.

WIDELINE_CONTROL

Supported Devices

As of this printing, The `WIDELINE_CONTROL gescape` is supported on the following graphics devices. Since this list may change, you should use `inquire_capabilities` to determine support for your device, as shown in the example that follows the description of this gescape.

- HP VMX with PowerShade
- Integrated Graphics Workstations with PowerShade
- Internal Color Graphics Workstations with PowerShade
- GRX with PowerShade
- CRX with PowerShade
- Dual CRX with PowerShade
- CRX-24 with PowerShade
- CRX-24Z
- CRX-48Z
- HP VISUALIZE-EG with PowerShade
- HCRX-8 with PowerShade
- HCRX-8Z
- HCRX-24 with PowerShade
- HCRX-24Z
- HP VISUALIZE-48
- HP VISUALIZE-48XP
- HP VISUALIZE-FX^2
- HP VISUALIZE-FX^4
- HP VISUALIZE-FX^6

Description

The ⟨*op*⟩ parameter is `WIDELINE_CONTROL`.

`WIDELINE_CONTROL` allows the application to use an implementation of widelines that is now supported by many device drivers. The implementation of widelines via `line_width`(3G) is restricted to 2D; however, this new implementation of widelines also works with `gopen` *mode* flag of `THREE_D` and Starbase's 3d vector primitives (e.g., `polyline3d`).

Two variations of these new widelines are now supported: filled widelines and stroked widelines. The main difference between these two is that filled widelines allow several kinds of endpoint styles to be specified, where stroked widelines are always "french-cut", i.e., they have vertical or horizontal ends. (See diagram below.)

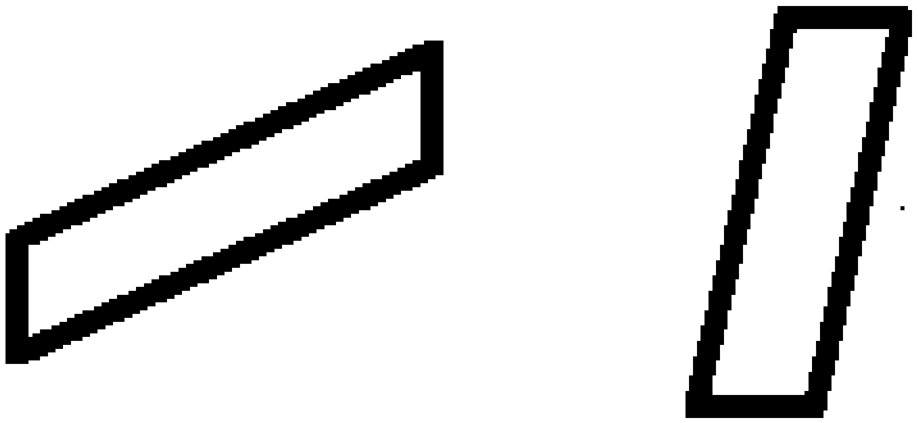

Figure 6-1. "French-Cut" Widelines

The `WIDELINE_CONTROL` `gescape` is intended to improve performance. On most graphics devices, stroked widelines will render faster than filled widelines. Also, both stroked and filled widelines will render faster than the 2D-only `line_width` style of widelines.

Attributes such as `line_type`, `line_color`, and modes such as `depth_cue` affect widelines just as they would affect normal-width aliased or anti-aliased lines. Anti-aliasing filled widelines is not supported. Anti-aliasing stroked widelines is not recommended, although the results may be satisfactory.

This `gescape` controls whether widelines are enabled (pixel width > 1), whether your system uses stroked or filled widelines, and what endpoint styles are used with the filled widelines.

Filled widelines are rendered by taking the single-pixel-width line segment and performing some calculations to generate a polygonal description of the wideline line segment. This polygonal description is then passed to the device's polygon rasterizer for rendering. Linetyped widelines require much more computation and are slower to render. Similarly, color per vertex and depth-cued vectors are slower to render.

Stroked widelines are rendered by taking the single-pixel-width line segment and replicating it horizontally or vertically, depending on its slope. Vectors whose major axis is horizontal will be replicated vertically (stacked atop one another); otherwise they will be replicated horizontally (stacked side-by-side). In either case, the outermost strokes are equidistant from the single-pixel-width line segment. Linetyped, color-per-vertex, or depth-cued modes of rendering will be significantly slower due to the complexity introduced with these modes.

Use `inquire_capabilities`(3g) to check the `IC_WIDELINE_CONTROL` bit in the `PRIMITIVES_2_CAPABILITIES` byte to determine if this gescape opcode is implemented on a particular `gopen` file descriptor.

Note that the computations for widening a line are only done in device coordinate XY space (screen space) based on the line width specified (in pixels); the Z coordinate is not changed. Thus, it displaces the rendered pixels after all modeling, viewing, and viewport transformations have been applied. A filled wideline's polygonal description will be clipped against the view frustrum; a stroked wideline's strokes are not re-clipped against the view frustrum but will be clipped to the current window.

This `gescape` requires that all of the data parameters described below are specified in the call.

The data parameters passed to the gescape entry point are used as follows (note that both *arg1* and *arg2* are used to pass information into the `gescape`):

`arg1.i[0]` The integer control flag:

- Set to −1 to not affect current wideline control setting
- Set to 1 to enable filled widelines
- Set to 2 to enable stroked widelines

`arg1.i[1]` The integer join style: (applies to filled widelines)

- Set to −1 to not affect current join style setting
- Set to 0 for butt join style
- Set to 2 for miter join style

`arg1.i[2]`	The integer cap style: (applies to filled widelines) ■ Set to −1 to not affect current cap style setting ■ Set to 0 for a butt cap style
`arg2.f[0]`	The floating point linewidth value. This is specified in pixels. Fractional values are legal but they are truncated to an integral value internally. Values are currently clamped to the range [1..16] for stroked widelines.

Note that cap styles may be extended in the future. Cap style is applied to the ends of a polyline; join style controls the appearance of the ends of segments within a polyline.

C Syntax Example

```
gescape_arg arg1, arg2;

arg1.i[0] = 1;
arg1.i[1] = 0;
arg1.i[2] = 0;
arg2.f[0] = 6.0;
gescape (fildes, WIDELINE_CONTROL, &arg1, &arg2);
```

FORTRAN77 Syntax Example

```
integer*4 arg1(64)
real  arg2(64)

arg1(1) = 1
arg1(2) = 0
arg1(3) = 0
arg2(1) = 6.0
call gescape(fildes, WIDELINE_CONTROL, arg1, arg2)
```

Pascal Syntax Example

```
{gescape_arg is defined in starbase.p1.h}

var
  arg1, arg2: gescape_arg;
begin
  arg1.i[1] = 1;
  arg1.i[2] = 0;
  arg1.i[3] = 0;
  arg2.f[1] = 6.0;
  gescape (fildes, WIDELINE_CONTROL, arg1, arg2);
```

Wideline vector performance and performance in general can be degraded by inefficient use of this gescape. The ability to specify −1 for the arguments allow some of the features of this gescape to be used without forcing the state associated with the rest of the gescape to be unnecessarily reset. However, the ideal usage of any modal or attribute setting is to set it sparingly around a batch of like-attribute, like-mode primitives.

The following is an example using this gescape.

```
int fildes;
unsigned char capabilities[SIZE_OF_CAPABILITIES];
int actual_size;
gescape_arg arg1, arg2;

:
/* Gopen the graphics device. */
fildes = gopen(...,THREE_D | MODEL_XFORM | INIT);


/* Inquire whether widelines are supported on this gopen context. */
actual_size = inquire_capabilities(fildes, SIZE_OF_CAPABILITIES, capabilities);

:
/* Set line color (magenta), line type, hidden surface, ... */

hidden_surface (fildes, FALSE, FALSE);
line_color (fildes, 1.0, 0.0, 1.0);
line_type (fildes, SOLID);

:
if (capabilities[PRIMITIVES_2_CAPABILITIES] & IC_WIDELINE_CONTROL)
{
   printf("WIDELINE_CONTROL supported by this device\n");

   /* Try to enable stroked widelines with width of 5 pixels. */

   arg1.i[0] = 2;
   arg1.i[1] = -1;
   arg1.i[2] = -1;
   arg2.f[0] = 5.0;

   gescape (fildes, WIDELINE_CONTROL, &arg1, &arg2);
}

/* Render polylines with or without move/draw flags. */

:
/* Change line_color to green */

line_color(fildes, 0.0, 1.0, 0.0);

/* Render polylines with or without move/draw flags. */

:
/* Change width to 8 pixels. */

arg1.i[0] = -1;
```

```
arg2.f[0] = 8.0;
gescape (fildes, WIDELINE_CONTROL, &arg1, &arg2);

/* Render polylines with or without move/draw flags. */
.
.
.
/* Change width to 1 pixel. */

arg1.i[0] = -1;
arg2.f[0] = 1.0;
gescape (fildes, WIDELINE_CONTROL, &arg1, &arg2);

/* Render polylines with or without move/draw flags. */
.
.
.
/* Change to filled widelines, butt for join and cap, width 7 */

arg1.i[0] = 1;
arg1.i[1] = 0;
arg1.i[2] = 0;
arg2.f[0] = 7.0;
gescape (fildes, WIDELINE_CONTROL, &arg1, &arg2);

/* Render polylines with or without move/draw flags. */
.
.
.
```

FINAL TRIM SIZE : 7.5 in x 9.0 in

A

Starbase Reference Pages

This appendix contains both new and updated reference pages for the HP-UX 10.X releases of Starbase.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Index