HP-UX Starbase Device Drivers Manual

Volume 1

HP 9000 Series 700 Computers

HEWLETT PACKARD

HP Part No. B2355-90047
Printed in USA April, 1993

Edition 1

Notices

Printing History

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

April 1993 . . . Edition 1. This manual is valid for HP-UX release 9.0 on all HP 9000 Series 700 Computers. This edition of the manual includes new HP VMX information as well as manual corrections.

This manual includes some Series 300/400/800 Starbase information; however, for revision 9.0 Starbase information on Series 300/400/800 computers, you should read the *HP-UX Starbase Device Drivers Manual* part number B2355-90019.

Contents

Figures

Tables

1

Introduction and Device Comparison

This manual documents the Starbase device drivers for the HP 9000 computers.

Manual Organization

This manual contains an introduction section, a section on developing a Starbase application which provides helpful techniques that are portable and work across a wide range of devices, and sections for each device driver and formatter with device-dependent information. The Portable Techniques section covers application development using either Xlib calls or Motif. The appendix contains the graphics escape (`gescape`) calls.

For each device, the following information is provided:

- Device Description—Key device features relative to using Starbase.
- Setting up the Device—Hardware and software configuration.
- Device Initialization—The device's default parameters, how to open the device in a program, etc.
- Starbase Functionality—How Starbase works on the device, what graphic escape sequences (gescapes) are provided, etc.

Other Useful Documents

The manual assumes that you understand the Starbase graphics library and have access to the following documents:

- *Introduction to Graphics*—This manual is a very low-level introduction to the most basic of computer graphics concepts. If you have never had experience with computer graphics, this manual is an excellent ice-breaker.
- *Beginner's Guide to Using Starbase*—This document is for those who know the basic concepts of computer graphics, but who have had no experience with Hewlett-Packard's Starbase graphics library.
- *Starbase Reference*—This manual is a small "brick" that covers only Starbase procedures. Included are syntax, semantics, and a discussion of functionality.
- *Starbase Pocket References*—These manuals, one each for C, FORTRAN, and Pascal, are pocket-sized references that cover only the procedures' names, types, and parameters. They are similar in content to the corresponding sections of the Quick Reference appendix in this manual, except it is in a convenient pocket-sized format. For a discussion of functionality above and beyond what the *Pocket References* have, see the *Starbase Reference*.
- *Starbase Display List Programmer's Manual*—This manual relates to Starbase Display List.
- *Starbase Graphics Techniques*—This 3-volume manual contains more in-depth explanations of Starbase routines and their use.
- Manuals provided with your graphics devices.

Introduction Chapter

The "Introduction and Device Comparison" has a number of tables to help you find and compare driver information. They contain the following information.

- The products that are supported for each device driver.
- The software revisions that affect each driver.
- The revisions that affect each formatter.
- The drivers supported on each computer.
- The graphics libraries that are supported on each window system.

The following tables provide a reference lists of supported products and device information to be used with them.

Important! You can have fast access to specific information about your graphics device (product name, device driver and capabilities supported) through the `graphinfo` utility installed with HP-UX 9.0 and later revisions. Simply type `/usr/bin/graphinfo`. For a detailed description of this utility, see the manpage on `graphinfo(1G)` in the *Starbase Reference Manual*.

Product Support Information

Table 1-1. Bit-Mapped Displays Product Support Information

Device Name	Device Path	Device Driver	Libraries Archive or Shared
hpgcrx Family			

HP GRX
HP CRX
HP Dual CRX
HP CRX-24
HP CRX-24Z
Integrated Grayscale[1]
Integrated Color[2]
	`/dev/crt`
	or
	`/dev/screen/`
	<dev_name>[3]
		`hpgcrx`	`libddgcrx.a`
			
			`libddgcrx.sl`

hpcrx48z			

HP CRX-48Z	`/dev/screen/`
	<dev_name>
		`hpcrx48z`	`libddcrx48z.a`
			
			`libddcrx48z.sl`

1 Integrated Grayscale includes the integrated graphics of the grayscale version of the Series 705, 710, 715, and 725.

2 Integrated Color includes the integrated graphics of the color version of the Series 705, 710, 715, and 725.

3 CRX and GRX will be supported in raw mode (using device path `/dev/crt`) until HP-UX 10.0.

Table 1-1.
Bit-Mapped Displays Product Support Information (continued)

Device Name	Device Path	Device Driver	Libraries Archive or Shared
HP Entry-Level VRX			
HP EVRX	`/dev/crt` or `/dev/screen/` <*dev_name*>	`hpevrx`	`libddevrx.a` `libddevrx.sl`
PersonalVRX P1 PersonalVRX P2 PersonalVRX P3	`/dev/crt`[1] or `/dev/screen/` <*dev_name*>	`hp98704` `hp98705`	`libdd98704.a` `libdd98705.a` `libdd98704.sl` `libdd98705.sl`

1 PersonalVRX will be supported in raw mode (using device path `/dev/crt`) until HP-UX 10.0.

Table 1-2. Bit-Mapped Displays Product Support Information

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

SRX	HP 98720A
	HP 98721
	HP 319SRX
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98720`
			`hp98721`
				`libdd98720.a`
				`libdd98721.a`
				
				`libdd98720.sl`
				`libdd98721.sl`

TurboSRX	HP 98730
	HP 98731
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98730`
			`hp98731`
				`libdd98730.a`
				`libdd98731.a`
				
				`libdd98730.sl`
				`libdd98731.sl`

TurboVRX T1
TurboVRX T2
TurboVRX T3
	HP 98735A
	HP 98736A
	HP 98736B
		`/dev/crt`[1]
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98735`
			`hp98736`
				`libdd98735.a`
				`libdd98736.a`
				
				`libdd98735.sl`
				`libdd98736.sl`

TurboVRX T2
TurboVRX T4
	HP 98765A
	HP 98766A
	X11
		`/dev/crt`[1]
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98765`
			`hp98766`
				`libdd98765.a`[2]
				`libdd98766.a`[2]
				
				`libdd98735.sl`
				`libdd98736.sl`
				`libdd98765.sl`[2]
				`libdd98766.sl`[2]

1 TurboVRX will be supported in raw mode (using device path `/dev/crt`) until HP-UX 10.0.

2 Available only on Series 700. Can be used interchangeably with the HP 98735/36 drivers on the Series700.

**Table 1-2.
Bit-Mapped Displays Product Support Information (continued)**

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

VRX Mono	HP A1096A
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hpa1096`	`libdda1096.a`
				
				`libdda1096.sl`

VRX Color	HP A1416A
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98550`	`libdd98550.a`
				
				`libdd98550.sl`

HP 300
Hi-Res
Display
	HP 318M
	HP 98544B
	HP 98545A
	HP 98547A
	HP 98549A
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp300h`	`libdd300h.a`
				
				`libdd300h.sl`

HP 300
Lo-Res
Display
	HP 98542A
	HP 98543A
	HP 310
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp300l`	`libdd300l.a`
				
				`libdd300l.sl`

C+
MH
C+
CH
	HP 319C+
	HP 98548A
	HP 98549A
	HP 98550A
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98550`	`libdd98550.a`
				
				`libdd98550.sl`

CHX	HP 98556A
	HP 98549A
	HP 98550A
	X11
		`/dev/crt`
		or
		`/dev/screen/`
		<*dev_name*>
			`hp98556`	`libdd98556.a`
				
				`libdd98556.sl`

Table 1-3. Input Devices Product Support Information

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared
HP-HIL Button Box	HP 46086A	`/dev/hilx` or `/dev/hilx_x`	`hp-hil`	`libddhil.a` `libddhil.sl`
HP-HIL Keyboard	HP 46020A HP 46021A	`/dev/hilx` or `/dev/hilx_x`	`hp-hil`	`libddhil.a` `libddhil.sl`
HP-HIL Knobs	HP 46083A HP 46084A	`/dev/hilx` or `/dev/hilx_x`	`hp-hil`	`libddhil.a` `libddhil.sl`
HP-HIL Mouse	HP 46060A HP 46060B HP 46095A with HP 46094A	`/dev/hilx` or `/dev/hilx_x`	`hp-hil`	`libddhil.a` `libddhil.sl`

Table 1-3. Input Devices Product Support Information (continued)

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

HP-HIL Tablets	HP 45911A
	HP 46087A
	HP 46088A
		`/dev/hilx`	`hp-hil`	`libddhil.a`
				
				`libddhil.sl`

HP-HIL Trackball	HP 80409A	`/dev/hilx`
		or
		`/dev/hilx_x`
			`hp-hil`	`libddhil.a`
				
				`libddhil.sl`

Keyboards	HP 46020A
	HP 46021A
	HP ASCII Terminals
		`/dev/tty`	`kbd` and `lkbd`	`libddkbd.a`
				
				`libddlkbd.a`
				
				`libddkbd.sl`
				
				`libddlkbd.sl`

Table 1-4. Graphics Hardcopy Product Support Information

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

HP-GL	HP 7440A
	HP 7470A
	HP 7475A
	HP 7550A
	HP 7570A
	HP 7575A
	HP 7576A
	HP 7580A/B
	HP 7585B
	HP 7586B
	HP 7595A
	HP 7596A
	HP 9111A
	HP C1600A[1]
	HP C1601A[1]
		`/dev/hpgl`	`hpgl`[2] or
			`hpgls`[3]
				`libddhpgl.a`
				`libdvio.a`
				
				`libddhpgl.sl`

HP-GL
CADplt
	HP 7510A
	HP 7550A
	HP 7570A
	HP 7575A
	HP 7576A
	HP 7580B[4]
	HP 7585B[4]
	HP 7586B
	HP 7595A
	HP 7596A
	HP C1600A[1]
	HP C1601A[1]
		`/dev/hpgl`	`CADplt`	`libddCADplt.a`
				
				`libddCADplt.sl`

1 Only in emulate mode

2 HP-GL devices with HP-IB interface.

3 HP-GL devices with RS-232 interface.

4 With HP-GL/2 plug-in cartridge.

Table 1-4.
Graphics Hardcopy Product Support Information (continued)

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

HP-GL/2 CADplt2	HP 7595B
	HP 7596B
	HP 7599A
	HP C1600A
	HP C1601A
	HP C1602A[1]
	HP C1620A
	HP C1625A
	HP C1627A
	HP C1629A
	HP C1631A[2]
		`/dev/hpgl2`	`CADplt2`	`libddCADplt.a`
				
				`libddCADplt.sl`

1 With HP-GL/2 plug-in cartridge.

2 This product is supported with these conditions: the printer is configured for a PCL-5 device, the only supported paper sizes are A (standard 8.5 x 11 inches) and A4 (the European equivalent of size A), and there is no gamma or color correction available in the CADplt2 driver for the PJ-XL300 dark hues. Color correction is left to the user.

Table 1-5. Other Output Drivers Product Support Information

Device Name	Supported Products	Device Path	Device Driver	Libraries Archive or Shared

HP-CGM			`hpcgm`	`libddhpcgm.a`
				
				`libddhpcgm.sl`

HP SBV[1]			`hpsbv`	`libddsbv.a`
				
				`libddsbv.sl`

Graphics Terminals	HP 150A
	HP 150 II
	HP 2393A
	HP 2397A
	HP 2623A
	HP 2627A
	HP 2625A
	HP 2628A
		`/dev/tty`	`hpterm`	`libddhpterm.a`
				
				`libddhpterm.sl`

1 Driver outputs a geometry metafile.

HP-UX Revision Support

Table 1-6.
Bit-Mapped Displays/Memory Drivers HP-UX Revisions

Device Name	Device Type	Series 300	Series 400	Series 700	Series 800
hpgcrx Family					
GRX	HP A1924A	—	8.0 - 9.0	8.01 - 9.01	—
CRX	HP A1659A	—	8.0 - 9.0	8.01 - 9.01	—
Dual CRX	HP A2269A	—	—	8.07 - 9.01	—
CRX-24	HP A1439A	—	—	8.07 - 9.01	—
CRX-24Z	HP A1454A	—	—	8.07 - 9.01	—
HP 705 Gray	HP A2289A	—	—	9.01	—
HP 705 HiRes	HP A2248A	—	—	9.01	—
HP 705 MedRes	HP A2249A	—	—	9.01	—
HP 715 Gray	HP A2610A	—	—	9.01	—
HP 715 HiRes	HP A2613A	—	—	9.01	—
HP 715 MedRes	HP A2612A	—	—	9.01	—
HP 710 Gray	HP A2208A	—	—	8.07 - 9.01	—
HP 710 HiRes	HP A2213A	—	—	8.07 - 9.01	—
HP 710 MedRes	HP A2210A	—	—	8.07 - 9.01	—
HP 725 HiRes	HP A2627A	—	—	9.01	—
HP 725 MedRes	HP A2626A	—	—	9.01	—

Table 1-6.
Bit-Mapped Displays/Memory Drivers HP-UX Revisions (continued)

Device Name	Device Type	Series 300	Series 400	Series 700	Series 800
hpcrx48z					
CRX-48Z	HP A2091A	—	—	9.01	—
HP Entry Level VRX					
HP EVRX	HP 425E HP 382	—	8.0 - 9.0	—	—
PersonalVRX **PersonalVRX**	HP 98704 HP 98705	7.03 - 9.0	7.03 - 9.0	8.01 - 9.01	—
SRX	HP 98720 HP 98721	5.18 - 9.0	—	—	1.1 - 8.0
TurboSRX	HP 98730	6.2 - 9.0	—	—	3.0 - 8.0
TurboVRX T1 **TurboVRX T2** **TurboVRX T3**	HP 98735[1] HP 98736A[2] HP 98736B[2]	7.03[3]	7.03 - 9.0[3]	[4]	—
TurboVRX T2 **TurboVRX T4**	HP 98765[1] HP 98766[2]	—	—	8.01 - 9.01	—
VRX Mono	HP A1096A	—	7.03 - 9.0	—	—
VRX Color	HP A1416A	—	7.03 - 9.0	—	—

1 Does not make use of accelerator.

2 Makes use of accelerator.

3 Only supported on selected models of Series 300 and Series 400.

4 The HP 98735/36 drivers may be used with the HP 98765A and HP 98766A products. The HP 98735A/36A/B physical devices are not supported on the Series 700.

Table 1-6.
Bit-Mapped Displays/Memory Drivers HP-UX Revisions
(continued)

Device Name	Device Type	Series 300	Series 400	Series 700	Series 800
MH	HP 98548A	5.5 - 9.0	—	—	—
C+	HP 98549A	5.5 - 9.0	—	—	—
CH	HP 98550A	5.5 - 9.0	—	—	1.2 - 8.0
CHX	HP 98556A	6.0 - 9.0	—	—	2.0 - 8.0
HP 300 Hi Res Display	HP 98544A/B 45A 47A	5.0 - 9.0	—	—	—
HP 300 Lo Res Display	HP 98542A HP 98543A	5.0 - 9.0	—	—	—
SOX11	Starbase on X11	6.5 - 9.0	7.03 - 9.0	8.01 - 9.01	3.0 - 9.0
HPVMX	HP Virtual Memory X	—	—	9.0 - 9.01	—

Table 1-7. Input Devices HP-UX Revisions

Device Name	Series 300	Series 400	Series 700	Series 800
Keyboards	5.0 - 9.0	7.03 - 9.0	8.01 - 9.01	1.0 - 9.0

Table 1-8. Graphics Hardcopy HP-UX Revisions

Device Name	Series 300	Series 400	Series 700	Series 800
HP-GL	5.0 - 9.0	7.03 - 9.0	8.01 - 9.01	1.0 - 9.0
HP-GL CADplt	6.0 - 9.0	7.03 - 9.0	8.01 - 9.01	2.0 - 9.0
HP-GL/2 CADplt2	7.0 - 9.0	7.03 - 9.0	8.01 - 9.01	7.0 - 9.0

Table 1-9. Other Output Drivers HP-UX Revisions

Device Name	Series 300	Series 400	Series 700	Series 800
HP-CGM	6.2 - 9.0	7.03 - 9.0	8.01 - 9.01	3.0 - 9.0
HP SBV	7.0 - 9.0	7.0 - 9.0	8.01 - 9.01	8.0
HP SBDL	6.2 - 9.0	7.0 - 9.0	8.01 - 9.01	1.0 - 9.0
HP Term	6.2 - 9.0	7.0 -9.0	8.01 - 9.01	1.0 - 9.0

Driver Compatibility with High-Level Starbase

Starbase drivers are developed in concert with a particular release of the high-level Starbase code; thus, compatibility between drivers and the high-level code is assured. In the future, however, new drivers may be released without re-releasing the high-level code. To permit determining high-level Starbase and driver compatibility, the code modules each contain a **revision number**. The revision numbers can be found by using the `what` command. The following is an example of how this call can be used:

```
$ what /usr/lib/libsb1.a
/usr/lib/libsb1.a:
        STARBASE HP-UX 9.0 A.09.00 921020 libsb1.a $Revision: 500.1.99.1 $
        STARBASE HP-UX 9.0 A.09.00 921020 libsga $Revision: 500.1.99.11 $
        STARBASE HP-UX 9.0 A.09.00 921020 libddvmx $Revision: 500.1.99.11 $
$ what /usr/lib/libsb2.a
/usr/lib/libsb2.a:
       GRM Library   HP-UX 8.0   A.01 Protocol PROTO_VER  FILE_VERSION
        STARBASE HP-UX 9.0 A.09.00 920811 libsb2.a $Revision: 500.1.1.1 $
$ what /usr/lib/libddhil.a
/usr/lib/libddhil.a:
       $Revision: 500.1.1.1 $   $Date: 92/06/03 19:47:39 $ libddhil.a
```

The following table indicates compatibility between the high level Starbase code and the Starbase driver code. The Starbase and Driver Revisions correspond to the following HP-UX Releases:

Note An "x" in the following tables indicates all versions of that number are applicable. Example: 50.x indicates 50.1, 50.2, etc.

Table 1-10. HP-UX Releases and Starbase Revision Levels

300		400		700		800	
Release	**Revision**	**Release**	**Revision**	**Release**	**Revision**	**Release**	**Revision**
5.0	28.1	7.03	350.1.50.x	8.01	400.1.50.x	1.0	48.x
5.1	39.1	7.05	364.1.2.x	8.05	401.1.1x	1.1	80.x
5.18	50.x	8.0	400.1.8.x	8.07	402.1.1x	1.2	83.x
5.2	65.x	9.0	500.1.1.x	9.0	500.1.1.x	2.0,2.1	120.x
5.3	65.1.1.x			9.01	500.1.99.x or	3.0	200.1.10.1
					500.1.100.x		
5.5	65.1.3.1					3.1	270.1.2.1
6.0	110.1					7.0	300.1.2.1
6.2	150.1.2.1					8.0	400.1.8.x
6.5	250.1.2.1					9.0	402.1.90
7.0	300.1.2.1						
7.03	350.1.x.1						
7.05	364.1.2.x						
8.0	400.1.8.x						
9.0	500.1.1.x						

Formatters and Release Levels

Table 1-11. Series 300 Formatters and Release Levels

Formatter Section	Libraries Archive/Shared	HP-UX Release	Starbase/Formatter Revision
HP Printer Control Language (PCL) with imaging extensions	`libfmtpcl.a`	5.2	65.x
		5.5	65.1.31
	`libfmtpcl.sl`	6.0	110.1
		6.2	150.1.2.1
		7.0	300.1.2.1
		7.03	350.1.50.x
		7.05	364.1.2.x
		8.0	400.1.8.x
		9.0	500.1.1.x

Table 1-12. Series 400 Formatters and Release Levels

Formatter Section	Libraries Archive/Shared	HP-UX Release	Starbase/Formatter Release
HP Printer Control Language (PCL) with imaging extensions	`libfmtpcl.a`	7.03	350.1.50.x
		7.05	364.1.2.x
	`libfmtpcl.sl`	8.0	400.1.8.x
		9.0	500.1.1.x

Table 1-13. Series 700 Formatters and Release Levels

Formatter Section	Libraries Archive/Shared	HP-UX Release	Starbase/Formatter Release
HP Printer Control Language (PCL) with imaging extensions	`libfmtpcl.a`	8.01	400.1.50.x
		8.05	401.1.1.x
	`libfmtpcl.sl`	8.07	402.1.1.x
		9.0	500.1.1.x
		9.01	500.1.99.x or 500.1.100.x

Table 1-14. Series 800 Formatters and Release Levels

Formatter Section	Libraries Archive/Shared	HP-UX Release	Starbase/Formatter Release
HP Printer Control Language (PCL) with imaging extensions	`libfmtpcl.a`	1.1	80.*x*
		1.2	83.*x*
	`libfmtpcl.sl`	2.0,2.1	120.*x*
		3.0	200.1.10.1
		7.0	300.1.2.1
		8.0	400.1.8.*x*
		9.0	500.1.1.*x*

Series 300 Starbase Device Drivers

Table 1-15. Bit-Mapped Displays

Device Name	Driver Type	318	319 C+	319 SRX	320	330	340	345	350	360	370	375
CH	HP 98550		●[1]		●	●	●	●	●	●	●	●
CHX	HP 98556		●[1]			●	●	●	●	●	●	●
HP 300	HP 300H, 300L	●[2]			●	●	●[2]	●	●	●	●	●
HP 9836A					●	●		●	●	●	●	●
SRX	HP 98720/21			●	●	●	●	●	●	●	●	●
TurboSRX	HP 98730/31						●	●	●	●	●	●
TurboVRX	HP 98735/36 (A/B)											●
VRX Color	HP A1416					●	●	●	●	●	●	●

1 HP 98549 only (HP 98556 is not supported.)

2 HP 300H monochrome only.

Note At the 7.0 release and thereafter, Starbase does *not* support the Series 310 CPU.

Series 400 Starbase Device Drivers

Table 1-16. Bit-Mapped Displays

Device Name	Driver Type	400t	400s	400dl	425dl	425t	425s	433s	425e
CH	HP 98550	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]
CHX	HP 98556								
CRX	HP A1659A	●[2]	●[2]	●	●	●[2]	●[2]	●[2]	●
GRX	HP A1924A	●[2]	●[2]	●	●	●[2]	●[2]	●[2]	●
VRX mono	HP A1096	●	●	●	●	●	●	●	
HP 98550	HPA1416	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]	●[1]
Entry Level VRX	HP EVRX								●
PersonalVRX	HP 98704/705	●	●			●	●	●	
TurboVRX	HP 98735/736		●				●	●	

1 Only with the HP A1416A display card.

2 The Series 400t requires an SGC adapter and the Series 400s should be ordered with the integrated CRX/GRX option or with the SGC connector option.

Series 700 Supported Graphics Devices

Table 1-17. Bit-Mapped Displays

Graphics Device Name	Model 705	Model 710	Model 715 33MHz	Model 715/725 50MHz	Model 720	Model 730	Model 735	Model 750	Model 755
CRX					●	●	●	●	●
Dual CRX					●	●	●	●	●
CRX-24			●	●	●	●	●	●	●
CRX-24Z			●	●	●	●	●	●	●
CRX-48Z				●			●		●
GRX					●	●	●		
Integrated Grayscale[1]	●	●	●	●					
Integrated Color[2]	●	●	●	●					
PersonalVRX					●	●	●	●	●
TurboVRX					●	●	●	●	●

1 Integrated grayscale supported in Series 700 Models 705, 710, 715 and 725.

2 Integrated color supported in both medium resolution (1024x768) and high resolution (1280x1024) configurations of the Series 700 Models 705, 710, 715 and 725.

Series 700 graphics use SGC data buses.

Series 800 Device Drivers

Table 1-18. Bit-Mapped Displays

Device Driver	Driver Type	815	825	835	840	850 - 855
SRX	HP 98720, -21	●	●	●	●	
TurboSRX	HP 98730, -31	●	●	●		
VRX color	HP 98550,-556	●	●	●		

Other Supported Device Drivers

The following device types are also supported on the Series 300, 400, 700 and 800 models listed. For a list of specific product numbers, please refer to the Product Support tables at the beginning of this chapter.

- Computer Graphics Metafile (CGM)
- HP-GL
- HP-GL, CADplt
- HP-GL/2, CADplt2
- HP-HIL
- HP-HIL Keyboards
- Memory (Virtual Frame Buffer)
- HP Graphics Terminals

Graphics Libraries Supported Within Windows

The following table shows which graphics libraries run in different window systems that are supported on the HP 9000 Series 300, Series 400, Series 700 and Series 800 workstations.

Table 1-19.
Graphics Libraries Supported in the Different Window Systems

Window Systems	Starbase and Starbase Display List	AGP/ DGL	HP-GKS	Xlib Graphics	HP PHIGS[1]	PEXlib[2]
Raw Mode[3]	Yes	Yes	Yes	No	Yes	No
X11	Yes, via the HPVMX and SOX-11 drivers or the Starbase Display Drivers	No	Yes, via the SOX-11 driver or the Starbase Display Drivers	Yes	Yes	Yes

1 HP-PHIGS not supported on Series 800.

2 PEXlib supported on Series 700 only.

3 Raw mode support on Series 700: 9.0 — CRX, PersonalVRX, and TurboVRX and on 10.0 — none

FINAL TRIM SIZE : 7.5 in x 9.0 in

2

Developing a Starbase Application

Overview

This chapter is intended for those who wish to develop a Starbase application program. Much of the emphasis in this chapter will be on developing Starbase 3D graphics applications. If your application does not require 3D graphics you may prefer to use Xlib instead of Starbase, but you may still find portions of this chapter helpful.

Note Source code for the applications listed in this chapter can be found in the `/usr/lib/starbase/demos` directory on your system if the `STAR-DEMO` fileset is installed. Refer to the `makefile` in that directory for details on how to compile and link these applications.

If you are not yet familiar with basic Starbase or X11 Window System concepts, please refer to the following manuals:

- *Starbase Graphics Techniques*
- *Starbase Reference*
- *Programming with Xlib*

Starbase is a flexible and powerful programming library which often offers the developer a variety of techniques to accomplish a particular graphics task. However, some techniques will work well only on certain types of graphics devices while other techniques are portable and will work across a wide range of devices.

In this chapter we will learn about these portable techniques as we develop several small Starbase applications:

- The first application, `portable_sb.c`, will use Xlib calls to create a window and handle user input and other events.
- Next, we will develop a functionally equivalent application, `motif_sb1.c`, using Motif instead of direct Xlib calls.
- Then, we will build upon the Motif application as we learn about additional portable techniques, `motif_sb2.c` and `motif_sb3.c` .
- Finally, we will learn more about differences between graphics devices so that you can make careful use of device-specific features if you wish.
- Guidelines for using Starbase Motif widgets are supplied at the end of this chapter.

Section One: Using Xlib in a Sample Application

Note The following detailed information is intended for those programmers who are used to programming at the Xlib level. If you are developing a new application and prefer a simplified approach to programming, turn to Section Two: "Using Motif in a Sample Application".

Our first sample application, `portable_sb.c`, will be simple but powerful enough to illustrate a variety of portable techniques. The application will create its own X window and render a simple 3D multi-colored cube and then allows you to use the mouse to rotate the cube.

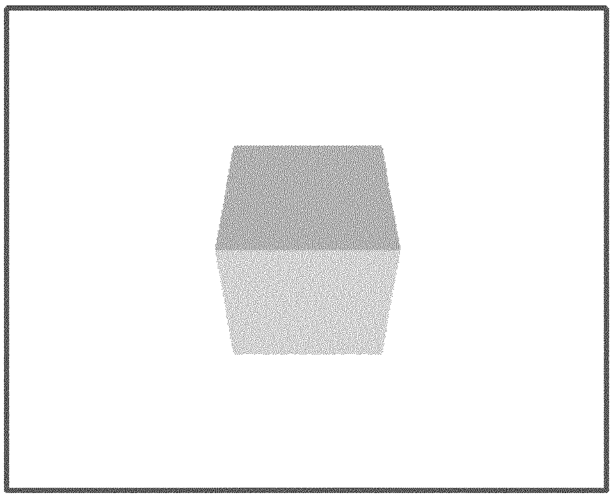

Figure 2-1. portable_sb window on a graphics device which can render solids.

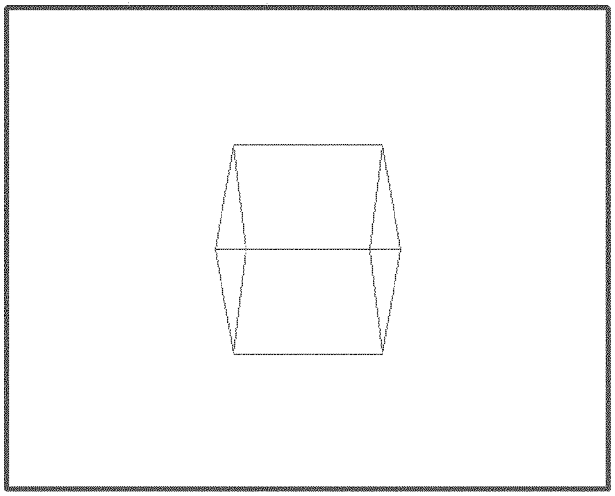

Figure 2-2.
portable_sb window on a graphics device which cannot render solids.

Let's look at the main steps the application must perform:

1. open the X display and select a screen
2. create an X window
3. gopen and map the X window
4. inquire capabilities of the graphics device
5. set display characteristics (shade mode, double buffering, etc)
6. handle X events, including user input
7. perform graphics rendering in response to events
8. [repeat steps 6 and 7 until exit]

In the following sections of this chapter, we will examine each step in detail and include code examples to show how to perform the step. Let's begin developing our application by showing the first several lines of code required to include header files and declare functions and variables used in the main function:

```
/*

  portable_sb.c - This file contains a complete, but simple, Starbase
  application. The application creates its own window and then renders a
  3D cube which may be rotated by moving the X pointer in the window with
  button 1 depressed. To exit the application, click pointer button 3.

*/
#include <starbase.c.h>
#include <stdio.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/XHPlib.h>
#include <math.h>
#include <wsutils.h>

/* Specify some #defines used later in the application */
#define DBUF_CAPABLE   0x01
#define SOLIDS_CAPABLE 0x02
#define WINDOW_NAME "WindowName"
#define ICON_NAME   "IconName"


/*

  main function

*/
main( argc, argv )
    int argc;
    char **argv;
{
    Window create_window();
    int gopen_and_map_window();
    int inquire_starbase_capabilities();
    void initialize_starbase();
    void handle_events();
    Display *display;
    int screen, width = 500, height = 400, screen_width, screen_height;
    int fildes, capabilities;
    Window window;
```

Step 1. Opening the X Display and Selecting a Screen

All HP graphics devices support Starbase in an X window. Older devices also allow Starbase access in raw mode (running without windows), but some newer devices do not support raw mode. To maximize portability across the family of HP graphics devices an application should create and use its own X window.

Before our Starbase application can perform any X Window operations, it must first open the X display. This is a simple operation which generally needs to be done only once, when the application first starts up:

```
/* Open the X display specified by the DISPLAY environment variable */
display = XOpenDisplay(NULL);

if ( display == NULL )
{
    if ( getenv("DISPLAY") == NULL )
        fprintf(stderr,"You need to set the DISPLAY env var\n");
    else
        fprintf(stderr,"Cannot open DISPLAY %s\n",getenv("DISPLAY"));
    exit(-1);
}
```

An X display will offer one or more screens but a Starbase application will generally run on the default screen of the display. There may be special situations in which an application may need to use other screens. See the *Programming with Xlib* manual for details on how to determine which screens are available on an X display. In this example the default screen of the display is used:

```
/* Use the default screen for the display */
screen = DefaultScreen( display );
```

Step 2. Creating an X Window

After our application has opened the X display and selected a screen, it can determine the width and height of the screen. Then, our application can proceed with the process of creating a window. To make the `main` function easier to understand, we'll create another function called `create_window` which will take the program argument and argument count, X display, X screen, and screen width and height as input parameters and return the window it creates.

```
/* Establish the width and height of the screen */
screen_width = DisplayWidth( display, screen );
screen_height = DisplayHeight( display, screen );

/* Create a window on the display and screen */
window = create_window( argc, argv, display, screen,
                       screen_width, screen_height );
```

There are several important points to remember when creating a window for a Starbase application:

1. The application should select the X visual most appropriate for the application.
2. If the application is not using the default visual, it must set at least the window colormap, background pixel color, and border pixel color attributes.
3. To make it easier to resize the window later, the application should first create the window with the largest possible size; then resize the window after the `gopen` but before it is mapped.
4. Some graphics devices do not support backing store. For maximum portability, the application should be prepared to detect expose events and redisplay all or part of the window as necessary.
5. Call `XSync` before `gopen`-ing the window.

In our sample application we use several utility functions contained in the `wsutils.c` file in the `/usr/lib/starbase/demos/SBUTILS` directory. These functions help to simplify our `create_window` function:

```
/*

  create_window - this function takes an X display and screen and returns
  the created window.

*/
Window create_window( argc, argv, display, screen,
                     screen_width, screen_height )
    int argc;
    char **argv;
    Display *display;
    int screen, screen_width, screen_height;
{
    int status, transparent_overlay_flag, depth, colormap_free_flag;
    int num_total_visuals, num_overlay_visuals, num_image_visuals;
    XVisualInfo *total_visuals, **image_visuals, *vis_info_ptr;
    OverlayInfo *overlay_visuals;
    Visual *visual_to_use;
    Colormap colormap;
    Window window;

    /* Get the list of all image visuals */
    if ( GetXVisualInfo( display, screen, &transparent_overlay_flag,
                        &num_total_visuals, &total_visuals,
                        &num_overlay_visuals, &overlay_visuals,
                        &num_image_visuals, &image_visuals ) != 0 )
    {
        fprintf(stderr,"Unable to find any visuals.\n");
        exit(-1);
    }

    /* Select the most appropriate visual for our application */
    if ( FindImagePlanesVisual( display, screen,
                               num_image_visuals, image_visuals,
                               CMAP_FULL, 24, FLEXIBLE,
                               &visual_to_use, &depth ) != 0 )
    {
        fprintf(stderr,"Unable to find an appropriate visual.\n");
        exit(-1);
    }
    /* Create the window (but don't map it yet) */
    if ( CreateImagePlanesWindow( display, screen,
                                  RootWindow(display,screen),
                                  0, 0, screen_width, screen_height,
                                  depth, visual_to_use, argc, argv,
                                  WINDOW_NAME, ICON_NAME, &window,
                                  &colormap, &colormap_free_flag ) != 0 )
    {
        fprintf(stderr,"Unable to create a window.\n");
        exit(-1);
    }

    /* Select several types of events on the window */
    XSelectInput( display, window,
          ExposureMask|StructureNotifyMask|ButtonPressMask|Button1MotionMask );

    /* Sync the create so that we can gopen() the window shortly */
    XSync( display, False );

    /* Return the created window to the calling function */
    return( window );

} /* end of create_window() */
```

Step 3. Gopening and Mapping the X Window

At this point, our application has opened the X display, selected a screen, and created a window. We are now ready to `gopen` the window and map it so that it will become visible. Once again, we will simplify the `main` function, this time by creating another function called `gopen_and_map_window` which will take several parameters. This function will perform the window gopen and map operations, and then return the Starbase file descriptor to the `main` function:

```
/* gopen() the window for Starbase graphics, and map the window */
fildes = gopen_and_map_window( argc, argv, display, window,
                               width, height );
```

Notice that we are passing the window width and height to this function. This is because the window was initially created to be full screen size (in pixels) to allow Starbase to easily handle resize events later, but before the window is mapped we want to specify the "real" size.

To ensure portability across the HP graphics device family, there are several important points to remember about `gopen`:

1. Link the application with Starbase shared libraries instead of archive libraries (see the chapter for your device driver in this manual for an explanation of how to do this).
2. Use the `NULL` constant as the driver parameter. This will allow Starbase to automatically choose the appropriate driver for the application.
3. An application should call `gopen` only once per window. Some devices allow more than one `gopen` per window, but using this feature will prevent the application from working properly on devices which support only one `gopen`.

With these points in mind, let us take a look at the first part of the `gopen_and_map_window` function:

```
/*

  gopen_and_map_window - this function takes the application arg count and
  arg values, the X display and window, and the intended window width and
  height.  This function gopens the window, establishes the width and
  height, and then returns the Starbase file descriptor.

*/
int gopen_and_map_window( argc, argv, display, window, width, height )
    int argc;
    char **argv;
    Display *display;
    Window window;
    int width, height;
{
    char *device, *list[1];
    int fildes;
    XSizeHints *size_hints;
    XClassHint *class_hint;
    XWMHints *wm_hints;
    XTextProperty window_name, icon_name;
    Atom wm_protocols[2];

    /* Create a device string based on the window id. */
    device = make_X11_gopen_string( display, window );

    /* Perform the gopen.  Use of the NULL driver name will allow Starbase to
       automatically decide which driver should be used.  The gopen flags
       are set for 3-D operation. */

    fildes = gopen( device, OUTDEV, NULL, INIT|THREE_D|MODEL_XFORM );

    if ( fildes < 0 )
    {
        /* Could not open the window. */
        fprintf(stderr,"Could not gopen window.\n");
        exit(-1);
    }
```

Now, we will establish the actual size of the window and set several window manager hints and protocols required for a “well behaved” X client:

```
    /* Now that the window has been gopened, we can establish the "real"
       size of the window. */

    XResizeWindow( display, window, width, height );
    /* We'll also set various properties used by the window manager. */

    list[0] = WINDOW_NAME;
    XStringListToTextProperty( list, 1, &window_name );
    list[0] = ICON_NAME;
    XStringListToTextProperty( list, 1, &icon_name );

    size_hints = XAllocSizeHints();
    size_hints->flags = USSize;
    size_hints->width  = width;
    size_hints->height = height;

    wm_hints = XAllocWMHints();

    class_hint = XAllocClassHint();
    class_hint->res_name = "portable_sb";
    class_hint->res_class = "Portable_sb";

    XSetWMProperties( display, window, &window_name, &icon_name,
                      argv, argc, size_hints, wm_hints, class_hint );

    /* Now, establish a list of protocols for communcation with the
       window manager.  The WM_DELETE_WINDOW protocol will cause the
       window manager to issue an event when the user selects "Close"
       from the window border menu.  The WM_SAVE_YOURSELF protocol will
       cause the window or session manager to issue an event when they
       plan to shut down. */

    wm_protocols[0] = XInternAtom( display, "WM_DELETE_WINDOW", False);
    wm_protocols[1] = XInternAtom( display, "WM_SAVE_YOURSELF", False);
    XSetWMProtocols( display, window, wm_protocols, 2 );
    XSetCommand( display, window, argv, argc );
```

Then, we can map the window and return the file descriptor to the calling function:

```
    /* Finally -- we can map the window!  We won't actually render anything
       to the window until the expose event happens later. */

    XMapWindow( display, window );

    /* Return the Starbase file descriptor to the calling function */
    return( fildes );

} /* end of gopen_and_map_window() */
```

Step 4. Inquiring Graphics Device Capabilities

idx|`inquire_capabilities`| After calling `gopen` on the window, an application may use Starbase inquiry calls to determine the device capabilities and its configuration. Through careful use of these inquiry calls an application can be designed to behave properly across the HP family of graphics devices. The *Starbase Reference* manual describes the complete set of inquiry functions. The `inquire_capabilities` and `inquire_fb_configuration` functions are especially useful for writing portable applications. In our sample application we will create another function of our own called `inquire_starbase_capabilities` and call it from the `main` function:

```
    /* Inquire the capabilities of the device */
    capabilities = inquire_starbase_capabilities( fildes );
```

A more sophisticated Starbase application might use such a function to check a variety of capabilities. In our sample application, we will use the function only to determine whether or not the device can support double buffering and rendering of 3D solids:

```
/*

   inquire_starbase_capabilities - this function takes the Starbase file
   descriptor and calls inquire_fb_configuration() and inquire_capabilities()
   to determine if the device can support double-buffering and rendering of
   solids.

*/
int inquire_starbase_capabilities( fildes )
    int fildes;
{
    int capabilities = 0;
    char sb_flags[ SIZE_OF_CAPABILITIES ];
    int image_banks, image_planes, planes_per_bank, overlay_planes;

    /* First, determine if the device and driver can do double-buffering */

    inquire_fb_configuration( fildes, &image_banks, &image_planes,
                              &planes_per_bank, &overlay_planes );

    if ( image_planes >= 16 )
        capabilities |= DBUF_CAPABLE;

    /* Then, verify that the device supports CMAP_FULL colormap mode and
       find out if the device renders solids */
```

```
        inquire_capabilities( fildes, SIZE_OF_CAPABILITIES, sb_flags );

        /* Verify that the device supports CMAP_FULL colormap mode */

        if ( ! ( sb_flags[ COLOR_1_CAPABILITIES ] & IC_CMAP_FULL ) )
        {
            fprintf(stderr,"Device does not support CMAP_FULL colormap mode.\n");
            exit(-1);
        }

        /* Find out if the device can render solids */

        if ( ( sb_flags[ HLHSR_CAPABILITIES ] & (IC_ZBUFFER_16|IC_ZBUFFER_24) )
          && ( sb_flags[ LIGHTING_CAPABILITIES ] & IC_LIGHTING )
          && ( sb_flags[ SHADING_CAPABILITIES ] & IC_SHADING ) )
        {
            /* We have the necessary capabilities to do 3D solids */
            capabilities |= SOLIDS_CAPABLE;
        }

        /* Return the capabilities flag to the calling function */
        return( capabilities );

    } /* end of inquire_starbase_capabilities() */
```

Device-dependent Starbase features are supported through the use of graphics escapes, or `gescapes`. By their very nature these gescapes are not portable and should be avoided in any application which is intended to run across the family of HP graphics devices. An application can determine whether some gescapes are supported or not on a device by using the information returned by `inquire_capabilities`. For example, the `IC_GAMMA_RENDERING` and `IC_GAMMA_CMAP` bits in the `COLOR_1_CAPABILITIES` flag indicate what type of gamma correction, if any, is available on a device. An application could use this information to determine whether or not to call the `GAMMA_CORRECTION` gescape.

Step 5. Setting Display Characteristics

After our application has determined the capabilities of the graphics device, we can set up the display characteristics. These characteristics will determine how subsequently rendered graphics primitives will appear to the user. Our sample application may not perform all the setup which your application requires so refer to the *Starbase Graphics Techniques* and *Starbase Reference* manuals for more information if necessary. We will create a function called `initialize_starbase` to do this setup work, and call the function from `main`:

```
/* Set up Starbase display characteristics */
initialize_starbase( fildes, width, height, screen_width, screen_height,
            capabilities );
```

Since the function performs a lot of different kinds of initialization tasks, we will examine the function in stages and explain each stage. The first steps are fairly portable across the family of graphics devices. We will:

- set a background color for our Starbase window,
- establish the device limits based on the actual window width and height,
- push an initial identity matrix on the stack (the matrix will be altered later to rotate the cube),
- specify an initial vertex format for later rendering of polygons. In our sample application, all polygons have the same coordinate data format so we only need to specify the vertex format once. If your application deals with polygons of various formats it can change the vertex format as the need arises.
- set up an initial view by calling our own function, `initialize_camera`. We will create the `initialize_camera` function after we have finished with this function.

```
/*

  initialize_starbase - this function performs various Starbase initialization
  tasks based on the window size and Starbase capabilities.

*/
void initialize_starbase( fildes, width, height, screen_width, screen_height,
                   capabilities )
    int fildes, width, height, screen_width, screen_height, capabilities;
{
    void initialize_camera();
    float width_fraction, height_fraction;
    static float identity_matrix[4][4] = {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0 };

    /* Establish the background color for window clears */

    background_color( fildes, 0.6, 0.8, 1.0 );

    /* Determine how to set p1 and p2 based on the actual window size
       compared to the size of the window at gopen() time */

    width_fraction = (float)width / (float)screen_width;
    height_fraction = (float)height / (float)screen_height;
    set_p1_p2( fildes, FRACTIONAL, 0.0, (1.0-height_fraction), 0.0,
              width_fraction, 1.0, 1.0 );

    /* Push an identity matrix on the stack for later use in rotating the
       cube based on user actions */

    push_matrix3d( fildes, identity_matrix );

    /* Establish an initial vertex format */

    vertex_format( fildes, 0, 0, 0, FALSE, CLOCKWISE );

    /* Specify an initial view camera */

    initialize_camera( fildes );
```

Next we will set the shade mode for our application. The shade mode establishes how our application will use colors and whether polygons and other primitives will be shaded or not. Our application will use direct color (Starbase `CMAP_FULL`) mode, but shading will be turned on only if the `inquire_starbase_capabilities` function reports that the device is capable of rendering solids. By **or**ing the `INIT` flag with our `CMAP_FULL` mode, we ensure that Starbase will properly initialize the colormap.

```
/* Now, we need to know about the ability of the device to do solids
   and double buffering. */

if ( capabilities & SOLIDS_CAPABLE )
    shade_mode( fildes, CMAP_FULL|INIT, TRUE );
else
    shade_mode( fildes, CMAP_FULL|INIT, FALSE );
```

If we determined that the device can double buffer graphics, we will turn it on at this point. Notice that we will request 24 planes per buffer, which may be more than the device can actually support. The `double_buffer` Starbase function will automatically adjust this requested value to the actual number of planes supported for the device.

Applications should generally avoid calling block operations such as `block_read` or `block_write` while double buffering is active. However, if this is necessary then the application can check the return value from the `double_buffer` function to determine exactly how many planes are used for double buffering. The application can use this information to reliably do block operations when double buffering is active. To see an example of how this can be done, refer to the "Block Operations" in the "Other Portable Techniques" section later in this chapter.

We then call the `dbuffer_switch` function to initialize double buffering to a known state which enables graphics rendering to buffer 0 and displays buffer 1. This allows all graphics devices to perform double buffering in a predictable way.

```
if ( capabilities & DBUF_CAPABLE )
{
    /* The device can do double buffering */
    double_buffer( fildes, TRUE|INIT, 24 );
    dbuffer_switch( fildes, 0 );
}
```

We complete the `initialize_starbase` function by looking at whether the device is capable of rendering solids or not. If the device can render solids, we turn on hidden surface removal and create a single light source. More sophisticated applications might use additional light sources, or might turn lights on or off interactively. If the device cannot render solids, we will render our polygons in "wireframe" mode by telling Starbase to make the polygons `INT_HOLLOW` with edges turned on.

```
        if ( capabilities & SOLIDS_CAPABLE )
        {
            /* We can do hidden surface removal and lighting and shading */
            interior_style( fildes, INT_SOLID, FALSE );
            clear_control( fildes, CLEAR_DISPLAY_SURFACE | CLEAR_ZBUFFER ) ;
            hidden_surface( fildes, TRUE, TRUE );
            light_source( fildes, 1, DIRECTIONAL, 1.0, 0.8, 1.0, 0.5, 1.0, 0.5 );
            light_switch( fildes, 0x2 );
        }
        else
        {
            /* We'll just draw the edges */
            interior_style( fildes, INT_HOLLOW, TRUE );
        }
    } /* end of initialize_starbase() */
```

Now that we have finished creating the `initialize_starbase` function, we will examine the `initialize_camera` function. This function is initially called by `initialize_starbase`, and then later whenever our window is resized. Our function will use the Starbase `view_camera` function to provide a simple camera model interface to viewing transformations. In this case, we want to point the camera at the cube which we'll be rendering later.

```
/*

  initialize_camera - this function is called to set up the Starbase
  view_camera() to point at the cube.

*/
void initialize_camera( fildes )
    int fildes;
{
    camera_arg camera;

    /* Specify an initial view camera */

    camera.refx = camera.refy = camera.refz = 0.0;
    camera.upy = 1.0;
    camera.upx = camera.upz = 0.0;
    camera.camx = 0.0;
    camera.camy = camera.camz = 5.0;
    camera.front = camera.back = 0.0;
    camera.projection = CAM_PERSPECTIVE;
    camera.field_of_view = 50.0;

    view_camera( fildes, &camera );

} /* end of initialize_camera() */
```

Step 6. Handling X Events

Note This step may not be necessary for very simple applications which do not wish to process user input or handle common X events such as resize and expose.

Most interactive applications, after initial setup has been completed, operate in a continuous loop that reads user input and then performs some action in response to the input. User input may consist of mouse movements, mouse button clicks, keyboard input, and so on. The application action in response to input might include rendering 2D or 3D graphics data.

Older Starbase applications which ran in raw mode (without windows) rely on Starbase input device capabilities and Starbase echos (cursors) to track user movement of the mouse or other input device. Starbase applications written to run in an X window, as described in this chapter, should use the X window input and cursor capabilities instead of Starbase calls.

An application which uses X input capabilities is usually event driven. That is, it receives an event, processes the event and responds to it, and then waits for the next event to occur. Events include user input and environment changes, such as exposing or resizing the window event, which could affect the application. While the application waits for an event, it is essentially dormant and not consuming system CPU cycles. The system CPU is then available to execute other applications.

In our sample application, we will deal with X events in a function called `handle_events` which will be called from the `main` function:

```
        /* Handle events and render graphics */
        handle_events( argc, argv, display, window, width, height,
                       screen_width, screen_height, fildes, capabilities );

    } /* end of main() */
```

We specified which X events we want to receive when we created the window in the `create_window` function. The `handle_events` function is responsible for actually reading events and responding appropriately to each event type. The function consists of a `while` loop which contains an `XNextEvent` call and a `switch` to handle the various event cases.

Without adding the detail of the switch cases, let's begin defining the function:

```
/*

  handle_events - this function takes information about the window and
  Starbase capabilities, and processes X events.  Starbase graphics
  (rendering the cube) is done in response to various events.
  The application is exited when the user clicks pointer button 3
  in the window.

*/
void handle_events( argc, argv, display, window, width, height,
                    screen_width, screen_height, fildes, capabilities )
    int argc;
    char **argv;
    Display *display;
    Window window;
    int width, height, screen_width, screen_height, fildes, capabilities;
{
    void draw_cube(), rotate_cube(), initialize_camera();
    XEvent event, next_event;
    float x_rotation, y_rotation, width_fraction, height_fraction;
    static int x_ptr_pos, y_ptr_pos;

    while ( TRUE )
    {
        /* Get the next X event.  This function will block until an event
           is received. */

        XNextEvent( display, &event );

        switch( event.type )
        {

                :
                :

        }
    }
} /* end of handle_events() */
```

Inside the `switch`, our function will handle several types of events. Let's first create the case to handle the Expose event. When the window is exposed, we first want to use the Starbase `clear_view_surface` function to clear the window. (This will also clear the Z-buffer if we are rendering solids). Then we will call our own `draw_cube` function to actually render our 3D cube. The `draw_cube` function is defined in the "Performing Graphics Rendering" section.

```
/* Expose events occur when the window is mapped, raised,
   de-iconified, and after it has been resized.  We want
   to render the cube in these cases.  Unless the "count"
   in the structure is 0, there are more expose events
   to follow and we might as well ignore this one. */

case Expose:
    if ( event.xexpose.count == 0 )
    {
        /* We must clear the view surface since the one done
           automatically by any previous dbuffer_switch() will
           have been lost. */
        if ( capabilities & DBUF_CAPABLE )
           clear_view_surface( fildes );

        /* Now, re-draw the cube. */
        draw_cube( fildes, capabilities );
    }
    break;
```

Next, we will create the case to handle `ConfigureNotify` events. These events can occur for lots of reasons but in our application we are only interested in resize events. So, we will first compare our "saved" window width and height to the actual window width and height. If no change has occurred, we'll ignore the event. If the size has changed, then we need to save the width and height for later comparisons. We will clamp the size to the initial screen size to avoid potential Starbase problems. Then, we will compute the ratios of the width and height of the newly resized window to the width and height of the screen (the original window size) and use this information in the Starbase `set_p1_p2` call. We must also call our own `initialize_camera` function after the `set_p1_p2` in order to reset the Starbase viewing characteristics.

```
/* ConfigureNotify events occur for a variety of reasons but we
   really only care about ones caused by a resize of the
   window. */

case ConfigureNotify:
    if ( width != event.xconfigure.width ||
        height != event.xconfigure.height )
    {
       /* If a resize occured, we must take the new width and
          height and set_p1_p2() to let Starbase know that the
          window size has changed. */

       width = event.xconfigure.width;
       height = event.xconfigure.height;
       if ( width > screen_width ) width = screen_width;
       if ( height > screen_height ) height = screen_height;
       width_fraction = (float)width / (float)screen_width;
       height_fraction = (float)height / (float)screen_height;
       hidden_surface( fildes, FALSE, FALSE );
       set_p1_p2( fildes, FRACTIONAL, 0.0,
                 (1.0-height_fraction), 0.0,
                 width_fraction, 1.0, 1.0 );
       initialize_camera( fildes );
       if ( capabilities & SOLIDS_CAPABLE )
           hidden_surface( fildes, TRUE, TRUE );
    }
    break;
```

Now, we'll create a case to handle the `ButtonPress` event. If this event occurs we will exit the application if button 3 was pressed. Otherwise we'll save the current pointer position so that we can see how much the position has changed when the first `MotionNotify` event is received. The difference between the positions will determine how much to rotate the cube.

```
/* We want to be able to handle button press events for two
   reasons:

   1) To allow the user to click and hold button 1 and move
      the pointer to rotate the cube.

   2) To allow the user to click button 3 to exit. */

case ButtonPress:
    if ( event.xbutton.button == Button3 )
    {
        /* Close Starbase, destroy the window, close the display,
           then exit. */
        gclose( fildes );
        XDestroyWindow( display, window );
        XCloseDisplay( display );
        exit(0);
    }
    else
    {
        /* Remember this initial position so that we can tell how
        far the user has moved the pointer when the next event
        occurs. */
        x_ptr_pos = event.xbutton.x;
        y_ptr_pos = event.xbutton.y;
    }
    break;
```

Then, we will handle the `MotionNotify` event. Relatively small mouse movements by the user can generate lots of `MotionNotify` events. In fact, MotionNotify events can be generated more quickly than we can draw the 3D cube. If we don't account for this fact in our application then the cube rotation and rendering will appear to "lag behind" the user's mouse movements and give the user the impression that our application is not very responsive. To avoid this problem, we will add some code to read and discard all but the last `MotionNotify` event in the X event queue. Then we will use the last event as the one of interest. After receiving the event, we will compute the amount of rotation based on how far the mouse has moved since the last `ButtonPress` or `MotionNotify` event. We will pass the rotation information to the `rotate_cube` function. (The `rotate_cube` function is not discussed in this chapter but is part of the `portable_sb.c` source file in the `/usr/lib/starbase/demos` directory.) Then, we will redraw the cube by calling the `draw_cube` function. Finally, we will save the current position of the pointer for our next comparison.

```
/* MotionNotify events occur when the user clicks and holds
   button 1 and moves the pointer.  When this happens, we
   want to use the pointer motion to apply a rotation to the
   cube and then re-render the cube. */

case MotionNotify:
    /* We need to get rid of any extra motion notify events which
       may have already accumulated.  Otherwise, these queued
       events will cause latent graphics rendering and give the
       user the impression that the application is not very
       interactive. */
    while ( XEventsQueued( display, QueuedAfterReading ) > 0 )
    {
        XNextEvent( display, &next_event );
        if ( next_event.type != MotionNotify )
        {
            XPutBackEvent( display, next_event );
            break;
        }
        event = next_event;
    }

    /* Now that we have the event, compute an X and Y rotation to
       be applied to the cube.  We'll base the rotation on how
       far the pointer has moved relative to the overall size of
       the window. */
    x_rotation = ((event.xbutton.y - y_ptr_pos)*180.0)/height;
    y_rotation = ((event.xbutton.x - x_ptr_pos)*180.0)/width;

    /* Apply the rotation and re-draw the cube */
    rotate_cube( fildes, x_rotation, y_rotation, 0.0 );
    draw_cube( fildes, capabilities );

    /* Remember the pointer position for the next event */
    x_ptr_pos = event.xbutton.x;
    y_ptr_pos = event.xbutton.y;
    break;
```

Finally, we will handle the `ClientMessage` event. An event of this type can occur in our application for two possible reasons:

1. The window manager wants to delete (or close) our window. In this case our application may wish to save current state information to a file which could be read the next time the application starts up. The application should then close Starbase, destroy the window, close the X display, and then exit.
2. The window or session manager plans to shut down and it wants our application to perform operations needed to save itself so that it can be restored later. Again, our application may wish to save current state information in a file. Our application must also call the `XSetCommand` function to let the session manager know that the application has saved its state. The window should not be destroyed in this case.

```
/* ClientMessage events will occur when the window manager wants
   to communicate a "delete window" or "save yourself" message */

case ClientMessage:

    if ( (Atom)event.xclient.data.l[0] ==
        XInternAtom( display, "WM_DELETE_WINDOW", False ) )
    {
        /* We are being asked by the window manager to gracefully
           shut down.  We'll close Starbase, destroy the window,
           and close the display, and then exit. */

        gclose( fildes );
        XDestroyWindow( display, window );
        XCloseDisplay( display );
        exit(0);
    }

    else if ( (Atom)event.xclient.data.l[0] ==
        XInternAtom( display, "WM_SAVE_YOURSELF", False ) )
    {
        /* The window manager has asked us to save ourselves.
           This would be a good place to save any state info
           in history files, etc.  Notice that we should NOT
           destroy the window in this case. */

        XSetCommand( display, window, argv, argc );
    }

    break;
```

Step 7. Performing Graphics Rendering

In the "Handling X Events" section we made several references to our `draw_cube` function. In our simple application, this function simply renders a 3D cube. Your application might use a much more complex function containing all the Starbase calls needed to render the visual components of the application.

Let's explore several graphics rendering issues as we define our small `draw_cube` function. The first part of the function defines the geometry of the cube which we wish to render and also declares a static `buffer` variable and initializes it to 0. This variable will be used in the `dbuffer_switch` call to specify which double buffering buffer is being written to.

```
/*

  draw_cube - this function renders a 3D cube, based on the capabilities
  of the device.

*/
void draw_cube( fildes, capabilities )
    int fildes, capabilities;
{
    static float cube_top[4][3] = {
        1.0, 1.0,-1.0,  -1.0, 1.0,-1.0,  -1.0, 1.0, 1.0,  1.0, 1.0, 1.0 };
    static float cube_bottom[4][3] = {
        1.0,-1.0, 1.0,  -1.0,-1.0, 1.0,  -1.0,-1.0,-1.0,  1.0,-1.0,-1.0 };
    static float cube_right[4][3] = {
        1.0,-1.0,-1.0,   1.0, 1.0,-1.0,   1.0, 1.0, 1.0,  1.0,-1.0, 1.0 };
    static float cube_left[4][3] = {
       -1.0,-1.0, 1.0,  -1.0, 1.0, 1.0,  -1.0, 1.0,-1.0,  -1.0,-1.0,-1.0 };
    static float cube_back[4][3] = {
       -1.0,-1.0,-1.0,  -1.0, 1.0,-1.0,   1.0, 1.0,-1.0,   1.0,-1.0,-1.0 };
    static float cube_front[4][3] = {
       1.0,-1.0, 1.0,    1.0, 1.0, 1.0,  -1.0, 1.0, 1.0,  -1.0,-1.0, 1.0 };
    static int buffer = 0;
```

Next, our application will check the `DBUF_CAPABLE` flag in our `capabilities` parameter and explicitly clear the view surface if double buffering is not being used (see the "Inquiring Graphics Device Capabilities" and "Setting Display Characteristics" sections). If double buffering is being used, the view surface is cleared automatically after each `dbuffer_switch` call is made and the `clear_view_surface` is not necessary at this point.

```
    if ( ! (capabilities & DBUF_CAPABLE) )
        clear_view_surface( fildes );
```

Now we can actually draw the cube. Each face of the cube will be a different color drawn using a `fill_color` and `polygon3d` call. In our simple application, we need to include the `fill_color` calls to make each face of the cube a different color. However, to achieve best performance an application should generally avoid making `fill_color` or other Starbase calls such as `vertex_format` unless the calls are necessary. Note that if our application is running on a device which is not `SOLIDS_CAPABLE` (see the "Inquiring Graphics Device Capabilities" and "Setting Display Characteristics" sections earlier) only the edges of the polygons will be drawn.

```
    fill_color( fildes, 0.0, 0.0, 1.0 );
    polygon3d( fildes, cube_top, 4, FALSE );
    fill_color( fildes, 0.0, 1.0, 0.0 );
    polygon3d( fildes, cube_bottom, 4, FALSE );
    fill_color( fildes, 0.0, 1.0, 1.0 );
    polygon3d( fildes, cube_left, 4, FALSE );
    fill_color( fildes, 1.0, 0.0, 0.0 );
    polygon3d( fildes, cube_right, 4, FALSE );
    fill_color( fildes, 1.0, 0.0, 1.0 );
    polygon3d( fildes, cube_front, 4, FALSE );
    fill_color( fildes, 1.0, 1.0, 0.0 );
    polygon3d( fildes, cube_back, 4, FALSE );
```

After the cube has been drawn, we need to switch buffers if double buffering is being used. The `buffer` variable is first toggled and then the `dbuffer_switch` call is made to actually change which buffer is being written to (and which one is being displayed). We'll finish off our `draw_cube` function by calling `make_picture_current` to output all our Starbase calls to the graphics device and wait for the graphics device to actually finish rendering. An interactive application such as this sample one generally needs to perform one `make_picture_current` call after any rendering done in response to user input or other events.

```
    if ( capabilities & DBUF_CAPABLE )
    {
        buffer = !buffer;
        dbuffer_switch( fildes, buffer );
    }
    make_picture_current( fildes );
} /* end of draw_cube() */
```

Note To achieve best performance an application should avoid extra `make_picture_current` calls. For example, it would be unnecessary and bad practice to insert a `make_picture_current` call after each `polygon3d` call in our `draw_cube` function. Your application performance may be improved by using `flush_buffer` instead of `make_picture_current`.

Section Two: Using Motif in a Sample Application

In this section we will enhance the sample application we developed in the previous section to use Motif and the X toolkit instead of Xlib for our window operations. This example, `motif_sb1.c`, will use the Starbase Motif widget.

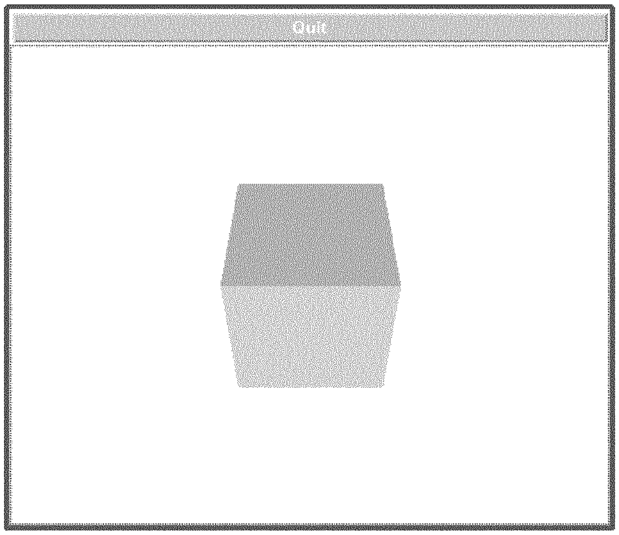

Figure 2-3. motif_sb1 window on a graphics device which can render solids.

The steps performed by our Motif application are fairly similar to the previous Xlib version:

1. initialize the X toolkit and create a widget hierarchy
2. create a Starbase Motif widget in the hierarchy
3. realize the widget hierarchy
4. inquire capabilities of the graphics device
5. set display characteristics (shade mode, double buffering, etc)
6. start the toolkit main loop and handle callbacks
7. perform graphics rendering in response to callbacks
8. [repeat steps 6 and 7 until exit]

In the following sections we will examine each step in detail and include code examples to show how to perform the step. Let's begin developing our Motif application by showing the first several lines of code required to include header files, and declare variables and functions used in the application:

```
/* motif_sb1.c - This file contains a complete, but simple, Starbase
   application.  The application uses the X toolkit, Motif, and the Starbase
   Motif widget to render a 3D cube which may be rotated by moving the X
   pointer in the window with button 1 depressed.  To exit the application,
   click on the "Quit" button.

*/
#include <stdio.h>
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Frame.h>
#include <Xg/Starbase.h>
#include <starbase.c.h>
#include <math.h>

/* Specify some #defines used later in the application */
#define DBUF_CAPABLE   0x01
#define SOLIDS_CAPABLE 0x02
#define EXPOSE_EVENT 1
#define RESIZE_EVENT 2
#define CLICK_EVENT  3
#define MOTION_EVENT 4
#define QUIT         5

/* Declare a context type variable which will be assigned in the main()
   function and used in the callback function. */
XContext context;

/* Create a structure to hold our context information */
typedef struct {
    int fildes;
    init capabilities;
    int x_ptr_pos;
    int y_ptr_pos;
} CONTEXT_DATA;
<newpage>

/*
  main function

*/
main( argc, argv )
    int argc;
    char **argv;
{
    void callback();
    int inquire_starbase_capabilities();
    void initialize_starbase();
```

```
Widget toplevel, form, quit_button, frame, sb;
XtAppContext app_context;
XmString quit_text;
Arg arg[15];
int n, fildes, capabilities;
static CONTEXT_DATA context_data;
```

Step 1. Initializing the X toolkit and Creating a Widget Hierarchy

Note For detailed information on Motif, please refer to the following manuals: *Mastering Motif Widgets*, and *HP OSF/Motif Programmer's Guide*.

The appearance and behavior of a Motif application's graphical user interface is determined in part by the widget hierarchy created by the application. In sophisticated applications this hierarchy can consist of many widgets of various types. Our simple application will create a very simple hierarchy containing a toplevel, form, pushbutton, frame, and Starbase widget:

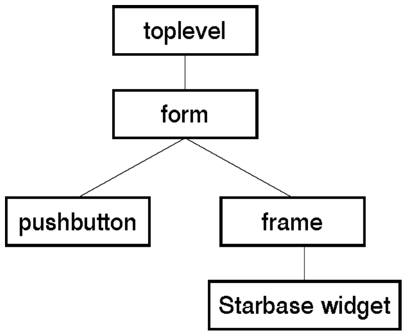

Figure 2-4. Sample Widget Hierarchy

A callback is often added to a widget when it is created. When the application adds a callback it specifies a function to be called and client data to be passed to the function when a particular event occurs. Callbacks are the primary mechanism by which events are handled in a Motif application. An application can have separate callback functions to handle all the different sorts of callbacks it must deal with, or it can consolidate all the callback handling into a single function. Our sample application will use the "single function" approach and use client data to tell the function which particular callback event is being handled.

Let's initialize the X toolkit and create all the components of this hierarchy except the Starbase widget. We'll add a callback to the "Quit" button so that our application can respond by exiting when the user clicks the button.

```
/* Initialize the toolkit and create the toplevel shell widget */

toplevel = XtAppInitialize( &app_context, "Motif", NULL, 0,
                           &argc, argv, NULL, NULL, 0);

/* Create a form widget to hold the rest of the children */

n = 0;
form = XtCreateManagedWidget( NULL, xmFormWidgetClass, toplevel, arg, n );

/* Create a quit button and add a callback so that our "callback"
   function will be invoked when the user clicks on the button.  We
   will pass a client data value of "QUIT" to the callback. */

n = 0;
XtSetArg( arg[n], XmNleftAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNrightAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNtopAttachment, XmATTACH_FORM ); n++;
quit_text = XmStringCreate( "Quit", XmSTRING_DEFAULT_CHARSET );
XtSetArg( arg[n], XmNlabelString, quit_text ); n++;
quit_button = XtCreateManagedWidget( "quit", xmPushButtonWidgetClass,
                                     form, arg, n );
XtAddCallback( quit_button, XmNactivateCallback, callback, QUIT );
XmStringFree( quit_text );

/* Create a frame to hold the Starbase widget -- just for decoration. */

n = 0;
XtSetArg( arg[n], XmNtopWidget, quit_button ); n++;
XtSetArg( arg[n], XmNtopAttachment, XmATTACH_WIDGET ); n++;
XtSetArg( arg[n], XmNleftAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNrightAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNbottomAttachment, XmATTACH_FORM ); n++;
frame = XtCreateManagedWidget( "frame", xmFrameWidgetClass,
                               form, arg, n );
```

Step 2. Creating a Starbase Motif Widget

Now that we've created the frame widget, we're ready to create a Starbase widget. In our application, we want to use the `CMAP_FULL` shade mode so we will specify that resource. Although our sample application also specifies the width and height of the Starbase widget, these resources should generally not be specified by the application itself. Instead, the width and height and other similiar size and position resources should be specified in an application resource file in `/usr/lib/X11/app-defaults` or in user-controlled resources. For details on how this can be done, see "Creating Default Files" in the *Mastering Motif Widgets* manual or the *HP OSF/Motif Programmer's Guide*.

```
/* Create the Starbase widget.  Use a shade mode of CMAP_FULL. */

n = 0;
XtSetArg( arg[n], XgNshadeMode, XgCMAP_FULL ); n++;
XtSetArg( arg[n], XmNwidth, 500 ); n++;
XtSetArg( arg[n], XmNheight, 400 ); n++;
sb = XtCreateManagedWidget( "Starbase", xgStarbaseWidgetClass,
                                                 frame, arg, n );
```

After creating the widget hierarchy, we need to add callbacks so that the user can interact with the application. Standard callbacks exist for expose, resize, and button click events, so we'll use those. Notice that we're passing different client data values such as `EXPOSE_EVENT` for each callback. This will enable our single callback function to distinguish between the various types of callbacks.

```
/* Add standard callbacks for expose, resize, and input (button click)
   so that our callback function will be invoked when these events occur.
   We'll pass client data values of EXPOSE_EVENT, RESIZE_EVENT, and
   CLICK_EVENT, respectively. */

XtAddCallback( sb, XmNexposeCallback, callback, EXPOSE_EVENT );
XtAddCallback( sb, XmNresizeCallback, callback, RESIZE_EVENT );
XtAddCallback( sb, XmNinputCallback,  callback, CLICK_EVENT );
```

We want to also allow our application to handle motion events so the user can click and hold the mouse button and move the mouse to rotate our 3D cube. Since there are not standard callbacks for this event we will use `XtAddEventHandler` to add a callback for the motion events:

```
/* Since there is no standard callback for motion events, we'll use
   XtAddEventHandler() to add a callback for those events. */

XtAddEventHandler( sb, Button1MotionMask, FALSE, callback, MOTION_EVENT );
```

Step 3. Realizing the Motif Heirarchy

After creating the widget hierarchy, we must realize the hierarchy. This step is somewhat analogous to mapping a window in our previous sample application, although the actual tasks performed by the toolkit when a hierarchy is realized are a bit more complicated. Our application can realize the toplevel widget and the toolkit takes care of realizing all the "children" of the toplevel, including the Starbase widget. When the Starbase widget is realized it will perform the Starbase `gopen` for us.

```
/* Realize the widgets (will also perform the Starbase gopen() for us).
   Note that our callback() function will be invoked when the Starbase
   widget is realized, but we will return without doing anything. */

XtRealizeWidget( toplevel );
```

Now that we have realized the Starbase widget, we can get the Starbase file descriptor from the widget. If the widget was unable to open the window, the file descriptor will have a value of -1.

```
/* Now that the Starbase widget has been realized, we can get the
   Starbase file descriptor. */

n = 0;
XtSetArg( arg[n], XgNfildes, &fildes ); n++;
XtGetValues( sb, arg, n);

if ( fildes < 0 )
{
    /* gopen() was not successful */
    fprintf(stderr,"Could not gopen window.\n");
    exit(-1);
}
```

Step 4. Inquiring Graphics Device Capabilities

After realizing the Starbase widget and obtaining the file descriptor, we can make several Starbase inquiries to determine what the graphics device is capable of and how it is configured. As we did in the first example application, we will simplify our `main` function by including these inquiries in another function of our own called `inquire_starbase_capabilities` and call it from the `main` function. The function itself is identical to the function defined in the first example.

```
/* Inquire the capabilities of the device */
capabilities = inquire_starbase_capabilities( fildes );
```

Step 5. Setting Display Characteristics

After our application has determined the capabilities of the graphics device, we can set up the display characteristics. Again, we will create a function called `initialize_starbase` and call it from `main`:

```
/* Set up Starbase display characteristics */
initialize_starbase( fildes, capabilities );
```

This function is virtually identical to the `initialize_starbase` function in the first example, although the Starbase `set_p1_p2` call is not needed in this function because this is done automatically by the Starbase widget. Refer to the `motif_sb1.c` source file in `/usr/lib/starbase/demos` directory for details of the `initialize_starbase` function.

Now that the initialization is done, we will create an X context for the window associated with the Starbase widget. This context will contain information such as the file descriptor that is specific to the widget. When our callback function is invoked it can then access this context data and use it to handle various types of events.

```
/* We will initialize the contents of a context structure and save the
   context with the Starbase window so that this information can later
   be obtained when callbacks occur. */
context_data.fildes = fildes;
context_data.capabilities = capabilities;
context = XUniqueContext();
XSaveContext( XtDisplay(sb), XtWindow(sb),  context, &context_data);
```

Step 6. Starting the Toolkit Main Loop and Handling Callbacks

This step is somewhat analogous to the event handling step in the first example. In this case, however, the X toolkit receives the actual X events, does some internal processing, and then invokes the application's callback functions if necessary. The toolkit will begin doing this after we start the toolkit main loop:

```
    /* Start the event handling loop */
    XtAppMainLoop( app_context );

} /* end of main() */
```

Now, we need to create our callback function which will be invoked by the toolkit when events occur. We'll begin by first defining the function parameters and declaring the variables used by the function:

```
/*

  callback - this function is invoked by the toolkit when widgets experience
  various events.  When we added the callbacks in main(), we specified
  client_data values for each callback so that we could identify the
  callbacks when they occurred.  For the CLICK_EVENT and MOTION_EVENT we
  are also interested in the call_data which will provide information about
  the X event which caused the callback.

*/
void callback( wdg, client_data, call_data )
    Widget wdg;
    caddr_t client_data;
    caddr_t call_data;
{
    void rotate_cube(), draw_cube(), initialize_camera();
    CONTEXT_DATA *context_data;
    XEvent *event;
    Dimension width, height;
    float x_rotation, y_rotation;
    Arg arg[10];
    int n;
```

After the variables have been declared we'll check to see if the callback is being invoked because the user clicked on the "Quit" button. If so, then we'll exit the application at this point.

```
    /* First check to see if the user hit the "Quit" button */
    if ( client_data == QUIT )
       exit(0);
```

If the callback was not invoked because of the "Quit" button, then it was invoked because of a Starbase widget event. In order to handle these events we need to find the context information we previously saved on the window associated with the Starbase widget. If the context has not yet been saved then we're not ready to handle the events and so our function will just return to the toolkit without doing anything.

```
        /* Attempt to find the window context data.  If it does not exist
           then our Starbase initialization has not been done and we don't want
           to do anything yet. */

        if ( XFindContext( XtDisplay(wdg), XtWindow(wdg),
                           context, &context_data ) != 0 )
            return;
```

After we have successfully found the context information, we can use a `switch` with different cases for each possible `client_data` value passed to the callback. Back in the `main` function we specified a different client data value for each callback added to the Starbase widget, so we can use the client data to determine exactly which callback event occurred. Let's first create the switch statement without the cases:

```
        /* The client_data will identify the type of callback which has just
           occurred. */

        switch( (int) client_data )
        {

                :
                :

        }

    } /* end of callback() */
```

Now, let's create a case for each possible `client_data` value. We'll begin with a case for the `EXPOSE_EVENT`. If an expose event occurs our callback needs to call the Starbase `clear_view_surface` function and then call our own `draw_cube` function. The `draw_cube` function is identical to the function created in our Xlib example. The file descriptor and capabilities are obtained from the context data we found earlier in the callback function.

```
            /* EXPOSE_EVENT occurs when the window is mapped, raised,
               de-iconified, and after it has been resized.  We want
               to render the cube in these cases. */
            case EXPOSE_EVENT:
                clear_view_surface( context_data->fildes );
                draw_cube( context_data->fildes, context_data->capabilities );
                break;
```

Next we'll create a case for the `RESIZE_EVENT`. The Starbase widget takes care of much of the work we did in our Xlib example. However, we do need to turn `hidden_surface` back on if our device is `SOLIDS_CAPABLE` because the Starbase widget's built-in resize handling will turn off hidden surface removal.

```
            /* RESIZE_EVENT occurs when the user resizes the window. We need to
               restore the hidden_surface() since the widget's built-in callback
               will have turned it off. */

            case RESIZE_EVENT:
                initialize_camera( context_data->fildes );
                if ( context_data->capabilities & SOLIDS_CAPABLE )
                    hidden_surface( context_data->fildes, TRUE, TRUE );
                break;
```

The `CLICK_EVENT` case is a little different from the previous two cases. We need to know the pointer position when the event occurred, so we must get this information from the X event which is included in a structure (`XmDrawingAreaCallbackStruct`) which is passed as the callback function's `call_data` parameter. The information is saved in the context data structure for use when the next `MOTION_EVENT` occurs.

```
/* CLICK_EVENT occurs when the user clicks a mouse button.  We want
   to remember the position of the pointer so that we can decide
   how much to rotate the cube when the user moves the pointer. */

case CLICK_EVENT:
    event = ((XmDrawingAreaCallbackStruct *) call_data)->event;
    context_data->x_ptr_pos = event->xbutton.x;
    context_data->y_ptr_pos = event->xbutton.y;
    break;
```

The `MOTION_EVENT` is the case which requires the most work in our example application. We need to know the current position of the pointer when the event occured. Since this callback was added using `XtAddEventHandler`, a pointer to the X event itself is passed to the callback function as the `call_data` parameter. Notice that we don't need to remove any extra motion events from the queue as we did in our Xlib example. This is because the X toolkit does this task automatically.

In addition to the current position of the pointer, we need to obtain the current width and height of the Starbase widget so we can scale our rotations based on the window size. We'll use the toolkit `XtGetValues` function to obtain this information. Then, we'll compute the cube's X and Y rotation based on the current and previous pointer position and the window size.

Next we'll apply the rotation by calling our own `rotate_cube` function and then re-render the cube by calling `draw_cube`. Finally we'll save the pointer position in the context data structure so that we can use it in the next `MOTION_EVENT` callback.

```
/* MOTION_EVENT occurs when the user clicks and holds
   button 1 and moves the pointer.  When this happens, we
   want to use the pointer motion to apply a rotation to the
   cube and then re-render the cube. */

case MOTION_EVENT:

    /* The call data is a pointer to the X event */
    event = (XEvent *) call_data;

    /* Determine the width and height of the widget */
    n = 0;
    XtSetArg( arg[n], XmNwidth, &width ); n++;
    XtSetArg( arg[n], XmNheight,&height ); n++;
    XtGetValues( wdg, arg, n );

    /* Compute an X and Y rotation to be applied to the cube.
       We'll base the rotation on how far the pointer has moved
       relative to the overall size of the window. */
    x_rotation =
        (event->xbutton.y - context_data->y_ptr_pos)*180.0/height;
    y_rotation =
        (event->xbutton.x - context_data->x_ptr_pos)*180.0/width;

    /* Apply the rotation and re-draw the cube */
    rotate_cube( context_data->fildes, x_rotation, y_rotation, 0.0 );
    draw_cube( context_data->fildes, context_data->capabilities );

    /* Remember the pointer position for the next event */
    context_data->x_ptr_pos = event->xbutton.x;
    context_data->y_ptr_pos = event->xbutton.y;
    break;
```

Step 7. Performing Graphics Rendering

The `draw_cube` function used in this Motif example is identical to the function used in the Xlib example. Please refer to that example for details about graphics rendering.

Section Three: Other Portable Techniques

In the previous two sections, we used a pair of example applications to learn some general concepts for developing Starbase applications which are portable across a variety of graphics devices. In this section we will build upon those examples and general concepts to learn about additional techniques which can be used to develop portable Starbase applications.

Block Operations

Block operations (`block_read`, `block_write`, and `block_move`) allow a Starbase application to directly access and manipulate the contents of the graphics device frame buffer. For general information about these functions refer to the *Starbase Reference* manual or the chapter called "Frame Buffer Control and Raster Opertions" in the *Starbase Graphics Techniques* manual.

Here, we will learn how to use one of these functions, `block_write`, in a portable way. You can then apply these same portable techniques to the use of the `block_read` function in your own application. The `block_move` function can be used directly without the need for these portable techniques. We will assume that our sample application, `motif_sb2.c`, needs to be able to write "blocks" of pixel data directly to the frame buffer. This pixel data will exist in our application in the form of three 8-bit arrays of red, green, and blue intensities.

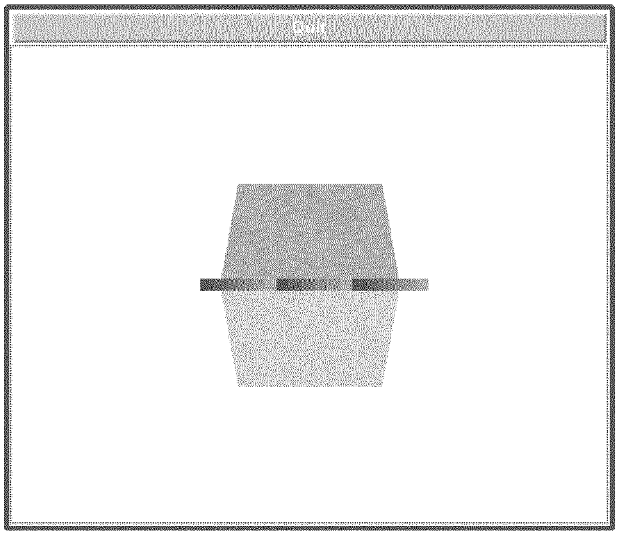

Figure 2-5. motif_sb2 window on a graphics device which can render solids.

Determining the Frame Buffer Depth

To use block operations, your application must know the depth used by Starbase for frame buffer operations. If your application is using double buffering then the depth is the return value from the `double_buffer` call. If double buffering is not used then the depth is the depth of the X window (or Starbase widget).

Let's make some minor changes to our Motif example application to enable the application to determine the Starbase frame buffer depth and record the depth for later use. First, we will define some new symbols, `DEPTH_24`, `DEPTH_12`, and `DEPTH_8`, along with our existing capabilities flags. These new symbols will be used to record the depth information in the capabilities flag word:

```
#define DBUF_CAPABLE   0x01
#define SOLIDS_CAPABLE 0x02
#define DEPTH_8        0x04
#define DEPTH_12       0x08
#define DEPTH_24       0x10
```

Next, we'll obtain the depth of the Starbase widget at the same time we obtain the file descriptor. This depth will be important if our application determines that the device cannot support double buffering.

```
    /* Now that the Starbase widget has been realized, we can get the
```

```
           Starbase file descriptor and widget depth. */

        n = 0;
        XtSetArg( arg[n], XgNfildes, &fildes ); n++;
        XtSetArg( arg[n], XmNdepth, &widget_depth ); n++;
        XtGetValues( sb, arg, n);
```

In our original example, our `inquire_starbase_capabilities` function determined the `capabilities` flag word contents. Now, we'll augment the capabilities flag word with depth information from our `initialize_starbase` function. To do that, we'll change the `initialize_starbase` function to type `int`, and pass the widget depth as an additional argument to the function. The function will then determine the actual Starbase frame buffer depth and return the `capabilities` flag word with that information recorded via the `DEPTH_8`, `DEPTH_12`, or `DEPTH_24` flag.

```
        /* Set up Starbase display characteristics */
        capabilities = initialize_starbase( fildes, capabilities, widget_depth );
```

Now, let's take a look at the changes to the `initialize_starbase` function. We've added the depth argument and declared an additional local variable, `starbase_depth`:

```
    int initialize_starbase( fildes, capabilities, window_depth )
        int fildes, capabilities, window_depth;
    {
        int starbase_depth;
```

If our device is capable of double buffering, we'll assign the return value from the `double_buffer` call to the `starbase_depth` variable. Otherwise, we'll assign the `window_depth` which was passed as an input argument.

```
        if ( capabilities & DBUF_CAPABLE )
        {
            /* The device can do double buffering */
            starbase_depth = double_buffer( fildes, TRUE|INIT, 24 );
            dbuffer_switch( fildes, 0 );
        }
        else
            starbase_depth = window_depth;
```

Then, we'll set either the `DEPTH_24`, `DEPTH_12`, or `DEPTH_8` flag in the `capabilities` flag word, based on the value of the `starbase_depth` variable. We'll return the capabilities flag word with this new flag set.

```
        if ( starbase_depth == 24 )
            capabilities |= DEPTH_24;
        else if ( starbase_depth == 12 )
            capabilities |= DEPTH_12;
        else
            capabilities |= DEPTH_8;
                :
                :
        return( capabilities );
```

At this point, our application has determined the Starbase frame buffer depth and has recorded it in the capabilities flag word. This depth information will be important when our application performs the `block_write` operation.

Writing the Pixel Data

The colors and intensities of pixels in our Starbase application X window (or Starbase widget) are determined by the contents of the frame buffer. The way in which the pixel appearance and frame buffer contents are related is determined by the Starbase colormap mode (set by `shade_mode`) and frame buffer depth. Our example application is using the Starbase `CMAP_FULL` colormap mode, which means that the Starbase colormap is initialized to a configuration which will provide a direct mapping between frame buffer contents and pixel colors.

Let's begin defining our own function, `rgb_block_write`, which will use the depth information contained in the capabilities flag word to determine how to write our application's red, green, and blue pixel data to the frame buffer. Our function will use the Starbase `dcblock_write` function to actually write the data.

```
    /*

      rgb_block_write - this function takes arrays of red, green, and blue data,
      converts it into 8-bit or 12-bit format if necessary, and then writes it
      to the specified x and y position.

      In this example function, the contents of the r, g, b input arrays are
      not modified.  In a real application it may be possible to use these
      input arrays to do the format conversion instead of allocating new
      arrays to hold the converted data.

    */
```

```
void rgb_block_write( fildes, capabilities, xpos, ypos, width, height,
                      r, g, b )
    int fildes, capabilities, xpos, ypos, width, height;
    unsigned char *r, *g, *b;
{
    unsigned char *rgb, *r12, *g12, *b12;
    int i;
```

First, we'll disable Starbase clipping so that we can freely block write anywhere in the X window.

```
    /* We'll turn off clipping so that we can do the block write anywhere in
       the X window */
    clip_indicator( fildes, CLIP_OFF );
```

Next, we need to know the frame buffer depth which was stored in the capabilities flag. If the frame buffer depth is 24, then there are three frame buffer banks: red (bank 2), green (bank 1), and blue (bank 0). Each bank can be considered an array of 8-bit data. To establish the color and intensity of a single pixel, the appropriate value (in the range 0..255) must be written to the corresponding location in each bank. So, in this case no conversion or modification of our input red, green, and blue data is necessary:

```
    if ( capabilities & DEPTH_24 )
    {
        /* No conversion is needed if we're working with a 24-bit frame
           buffer.  However, the bank_switch() calls are needed to
           let Starbase know which of the 3 frame buffer banks (red, green,
           or blue) we wish to write to.  Bank 2 is red, bank 1 is green,
           and bank 0 is blue. */

        bank_switch( fildes, 2, 0 );
        dcblock_write( fildes, xpos, ypos, width, height, r, FALSE );
        bank_switch( fildes, 1, 0 );
        dcblock_write( fildes, xpos, ypos, width, height, g, FALSE );
        bank_switch( fildes, 0, 0 );
        dcblock_write( fildes, xpos, ypos, width, height, b, FALSE );
    }
```

If the frame buffer depth is 12, we still have the red, green, and blue banks. Again, each bank can be considered an array of 8-bit data. In this case, however, there are only 4 bits of meaningful information per pixel. Whether this information is contained in the most significant or least significant 4 bits of the 8-bit data depends upon a variety of factors. So, when preparing an 8-bit value to be written it is best to ensure that the most significant 4 bits are duplicated in the least significant 4 bits. When the write occurs Starbase will ensure that only the appropriate half of the actual frame buffer data will be modified.

```
else if ( capabilities & DEPTH_12 )
{
    /* We need to convert to 12-bit values.  In order to preserve the
       input data, we'll allocate a local buffer to hold the converted
       data. */

    r12 = (unsigned char *) malloc( width*height );
    g12 = (unsigned char *) malloc( width*height );
    b12 = (unsigned char *) malloc( width*height );
    if ( r12 == NULL || g12 == NULL || b12 == NULL )
    {
        fprintf(stderr,"Cannot allocate memory\n");
        return;
    }

    /* In this case, we will duplicate the most significant nibble
       (4 bits) of each red, green, and blue value into the least
       significant nibble. */

    for ( i = 0; i < (width*height); i++ )
    {
        r12[i] = ( r[i] & 0xf0 ) | ( r[i] >> 4 );
        g12[i] = ( g[i] & 0xf0 ) | ( g[i] >> 4 );
        b12[i] = ( b[i] & 0xf0 ) | ( b[i] >> 4 );
    }

    /* Now that we have converted the data to 12-bit format, we need to
       write it to the window.  The bank_switch() calls are needed to
       let Starbase know which of the 3 frame buffer banks (red, green,
       or blue) we wish to write to.  Bank 2 is red, bank 1 is green,
       and bank 0 is blue. */

    bank_switch( fildes, 2, 0 );
    dcblock_write( fildes, xpos, ypos, width, height, r12, FALSE );
    bank_switch( fildes, 1, 0 );
    dcblock_write( fildes, xpos, ypos, width, height, g12, FALSE );
    bank_switch( fildes, 0, 0 );
    dcblock_write( fildes, xpos, ypos, width, height, b12, FALSE );

    /* Free our local buffers */
    free( r12 ); free( g12 ); free( b12 );

}
```

If the frame buffer depth is 8, we have only a single bank of 8-bit data. This case is a bit more complicated because, for each pixel, we need to "pack" the 24 bits of red, green, and blue data into a single 8-bit value. If the `SB_X_SHARED_CMAP` environment variable is set, we must pack the data using a 6|6|6 scheme. Otherwise, we must pack the data using a 3:3:2 scheme. See the "CRX Family of Device Drivers" chapter of this manual for detailed information about the 6|6|6 and 3:3:2 schemes. In our example, we will define a couple of macros to simplify conversion to 6|6|6 and 3:3:2.

```
        else if ( capabilities & DEPTH_8 )
        {
            /* We need to pack the 24-bit data into 8-bit values.  In order to
               preserve the input data, we'll allocate a local buffer to hold
               the converted data. */

            rgb = (unsigned char *) malloc( width*height );
            if ( rgb == NULL )
            {
                fprintf(stderr,"Cannot allocate memory\n");
                return;
            }

            /* There are two possible 8-bit formats, commonly known as 3:3:2 and
               6|6|6.  If the SB_X_SHARED_CMAP environment variable is set, we
               will use the 6|6|6 format.  Otherwise, we use the 3:3:2 format.
               We'll define some macros to make this easier. */

    #define RGB_TO_332( r, g, b ) ( (  r       & 0xe0 ) + \
                                    ( (g >> 3) & 0x1c ) + \
                                    ( (b >> 6) & 0x03 ) )

        /* RGB_TO_666_FACTOR is 6.0/256.0, which is needed to quantize a value in
           the range 0..255 into a range of 0..5 */

    #define RGB_TO_666_FACTOR 0.023438

    #define RGB_TO_666( r, g, b ) ( 40 + \
                                    (unsigned char)(r * RGB_TO_666_FACTOR) * 36 + \
                                    (unsigned char)(g * RGB_TO_666_FACTOR) * 6 + \
                                    (unsigned char)(b * RGB_TO_666_FACTOR) )
            if ( getenv("SB_X_SHARED_CMAP") )
            {
                for ( i = 0; i < (width*height); i++ )
                    rgb[i] = RGB_TO_666( r[i], g[i], b[i] );
            }
            else
            {
                for ( i = 0; i < (width*height); i++ )
                    rgb[i] = RGB_TO_332( r[i], g[i], b[i] );
            }

            /* Now that the data has been converted, we will write the 8-bit
               values into the window.  There is no need for a bank_switch()
               since we know that the appropriate 8-bit bank is already
               enabled for writing. */

            dcblock_write( fildes, xpos, ypos, width, height, rgb, FALSE );

            /* Free up our local buffer. */
            free( rgb );

        }
```

After we have written the data to the frame buffer, we'll restore the clip_indicator to the default value of `CLIP_TO_RECT` and return.

```
        /* Restore the clip_indicator() back to its default value */
        clip_indicator( fildes, CLIP_TO_RECT );

    } /* end of rgb_block_write() */
```

Transparent Overlay Windows

Applications generally perform graphics rendering in the graphics device image planes. Certain Hewlett-Packard graphics devices provide both image planes and overlay planes. On these devices, an application may wish to create a window in the overlay planes and make its background transparent so that a user can "see through" the overlay window to the image window beneath it.

Graphics rendering to the overlay window will not affect the image plane window contents, so an application can render text and other graphics to the overlay window without needing to re-render the image window graphics. This is especially useful if the image window contains very complex graphics which take a long time to re-render.

There are several important points to consider about the use of windows in overlay planes:

1. An application needs to be prepared to use an alternate approach on graphics devices which do not provide overlay planes. (Note that windows in the image planes cannot be made transparent).
2. Starbase provides reduced functionality in an overlay window. The `CMAP_FULL` colormap mode is not supported in an overlay window on all graphics devices, so an application should use `CMAP_NORMAL` instead. 3D hidden surface removal, lighting and shading are generally not supported in an overlay window. Double buffering is possible on some devices but not recommended.
3. An overlay window should use the default X colormap whenever possible. If this is not done, other X clients such as terminal windows may exhibit "color flashing" effects when the overlay window is focused by the window manager. Using the default X colormap means that selection of colors may be limited and colors must be specifically allocated to ensure that they will not be changed by other X clients.

In this section, we will modify our original Motif example to use a transparent overlay widget to display some text which will appear to sit "on top" of the 3D cube object in the image planes widget. If the application, `motif_sb3.c`, can create the overlay widget, the text will only need to be rendered when expose and resize events occur—the rotation and re-rendering of the 3D cube will not affect the text. Otherwise, the application will have to re-render the text every time the cube is re-rendered.

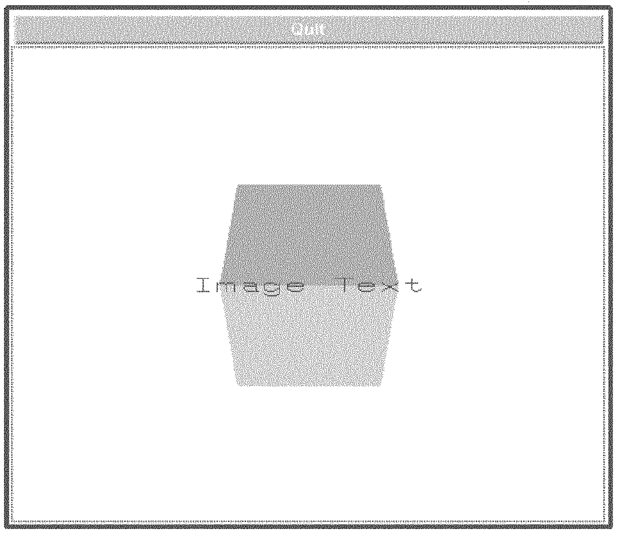

Figure 2-6. motif_sb3 window with solids rendering/no overlay planes.

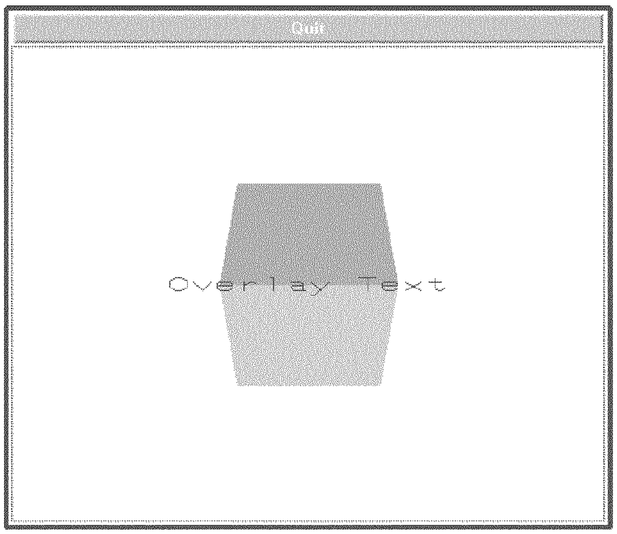

Figure 2-7. motif_sb3 window with solids rendering and overlay planes.

Creating the Overlay Starbase Widget

In our original example in the `main` function, we created a single Starbase widget which, by default, is placed in the graphics device image planes. In this example, we will create a Starbase widget in the overlay planes *before* we create the image planes widget. By creating the overlay planes widget first, we ensure that it will sit "on top" of the image planes widget. Notice that we are again creating a frame widget to hold the Starbase widget, and the frame is attached to its form widget parent in the same way as the frame for the image planes widget.

```
/* Create a frame to hold the overlay Starbase widget -- just for
   decoration. */

n = 0;
XtSetArg( arg[n], XmNtopWidget, quit_button ); n++;
XtSetArg( arg[n], XmNtopAttachment, XmATTACH_WIDGET ); n++;
XtSetArg( arg[n], XmNleftAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNrightAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNbottomAttachment, XmATTACH_FORM ); n++;
frame = XtCreateManagedWidget( "frame", xmFrameWidgetClass,
                               form, arg, n );
```

To create the overlay widget we must specify several resources. We will use the `XgCMAP_NORMAL` `shadeMode` and set the overlay resource to `True`. We wish to use the default X colormap so we must specify an `openMode` which excludes the `gopen INIT`. The `wmCmap` resource must be set to `XgWM_CMAP_LOW_PRIORITY` so that the window manager will properly focus the image widget colormap. Finally, we will use the `sox11` Starbase driver instead of the default device driver. Refer to the `XgStarbase(3X)` man page in the *Starbase Reference* manual for details about all of these resources.

```
/* Now create the overlay widget itself */

n = 0;
XtSetArg( arg[n], XgNshadeMode, XgCMAP_NORMAL ); n++;
XtSetArg( arg[n], XgNoverlay, True ); n++;
XtSetArg( arg[n], XgNopenMode, XgTHREE_D|XgMODEL_XFORM ); n++;
XtSetArg( arg[n], XgNwmCmap, XgWM_CMAP_LOW_PRIORITY ); n++;
XtSetArg( arg[n], XgNdriver, "sox11" ); n++;
XtSetArg( arg[n], XmNwidth, 500 ); n++;
XtSetArg( arg[n], XmNheight, 400 ); n++;

sb_overlay = XtCreateManagedWidget( "Starbase", xgStarbaseWidgetClass,
                                    frame, arg, n );
```

Verifying the Widget

After we create the overlay plane widget, we will proceed to create the image plane widget just as we did in our original example.

```
/* Create a frame to hold the image Starbase widget -- just for
   decoration. */

n = 0;
XtSetArg( arg[n], XmNtopWidget, quit_button ); n++;
XtSetArg( arg[n], XmNtopAttachment, XmATTACH_WIDGET ); n++;
XtSetArg( arg[n], XmNleftAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNrightAttachment, XmATTACH_FORM ); n++;
XtSetArg( arg[n], XmNbottomAttachment, XmATTACH_FORM ); n++;
frame = XtCreateManagedWidget( "frame", xmFrameWidgetClass,
                              form, arg, n );

/* Now create the image widget itself */

n = 0;
XtSetArg( arg[n], XgNshadeMode, XgCMAP_FULL ); n++;
XtSetArg( arg[n], XmNwidth, 500 ); n++;
XtSetArg( arg[n], XmNheight, 400 ); n++;
sb = XtCreateManagedWidget( "Starbase", xgStarbaseWidgetClass,
                           frame, arg, n );
```

Then, we'll add the resize and expose callbacks, but in this example we'll not add the input or button motion callbacks just yet. Instead, we'll go ahead and realize the widget heirarchy and then check the image file descriptor and perform the `inquire_starbase_capabilities` and `initialize_starbase` function calls just as before.

```
/* We want to add these callbacks to the image plane widget whether we
   were able to create a transparent overlay widget or not. */

XtAddCallback( sb, XmNresizeCallback, callback, RESIZE_EVENT );
XtAddCallback( sb, XmNexposeCallback, callback, EXPOSE_EVENT );

/* Realize the widgets (will also perform the Starbase gopen() for us).
   Note that our callback() function will be invoked when the Starbase
   widget is realized, but we will return without doing anything. */

XtRealizeWidget( toplevel );

/* Now that the image Starbase widget has been realized, we can get the
   Starbase file descriptor. */

n = 0;
XtSetArg( arg[n], XgNfildes, &fildes ); n++;
```

```
        XtGetValues( sb, arg, n);

        if ( fildes < 0 )
        {
            /* gopen() was not successful */
            fprintf(stderr,"Could not gopen window.\n");
            exit(-1);
        }

        /* Inquire the capabilities of the device */
        capabilities = inquire_starbase_capabilities( fildes );

        /* Set up Starbase display characteristics */
        initialize_starbase( fildes, capabilities );
```

Next, we must get information from the overlay widget which will indicate whether or not the widget resides in the overlay planes and can be made transparent. If we were successful, the transparent resource will hold the colormap index which can be used as a transparent color. Otherwise, the resource will have a value of -1.

```
        /* Next, we'll see if we were able to create the overlay window
           we had hoped for */

        n = 0;
        XtSetArg( arg[n], XgNfildes, &overlay_fildes ); n++;
        XtSetArg( arg[n], XgNtransparent, &transparent_index ); n++;
        XtGetValues( sb_overlay, arg, n);
```

If the overlay file descriptor is valid and there is a transparent colormap index, we can proceed with our plans to use the overlay widget. We will set the background color of the widget to the `transparent_index`. Whenever necessary, the X toolkit will automatically clear the widget background to this color—no Starbase `clear_view_surface` will be needed. Next we'll add callbacks for expose and resize events. Since the overlay widget will always sit "on top" of the image widget, we must establish the input and button motion callbacks for the overlay widget. Since we're planning to render Starbase text into the overlay widget, we will perform some text initialization tasks. We'll create our own `allocate_overlay_color` function so that we can allocate a color for the text, and then set the text color using the Starbase `text_color_index` call. Then, we'll create another function called `initialize_starbase_text` to complete the text initialization task. Finally, we'll `or` an `OVERLAY` flag into the capabilities flag word to indicate that we did successfully create an overlay widget.

NOTE: The `allocate_overlay_color` and `initialize_starbase_text` functions will be defined later.

```
if ( overlay_fildes >= 0 && transparent_index >= 0 )
{
    /* We were able to successfully gopen an overlay window which
       supports transparency.  Next, we'll establish the background
       color of the overlay widget to be transparent.  By doing so,
       we'll avoid the need to do a Starbase clear_view_surface(). */

    XtSetArg( arg[0], XmNbackground, transparent_index );
    XtSetValues( sb_overlay, arg, 1 );

    /* Then, we'll add callbacks to the overlay widget.  We need callbacks
       for the expose and resize events.  And, since our overlay widget
       will sit "on top" of the image widget, the overlay widget must
       be responsible for handling the user's input events.  So we'll
       need to set up callbacks for the button click and motion. */

    XtAddCallback( sb_overlay, XmNexposeCallback, callback, EXPOSE_EVENT );
    XtAddCallback( sb_overlay, XmNresizeCallback, callback, RESIZE_EVENT );
    XtAddCallback( sb_overlay, XmNinputCallback,  callback, CLICK_EVENT );
    XtAddEventHandler( sb_overlay, Button1MotionMask, FALSE, callback,
                      MOTION_EVENT );

    /* Finally, we'll perform some Starbase initialization on the
       overlay widget (mainly setting up text characteristics) */

    color_index = allocate_overlay_color(sb_overlay, 1.0, 0.0, 0.0);
    text_color_index( overlay_fildes, color_index );
    initialize_starbase_text( overlay_fildes );

    /* Set a flag in capabilities which will indicate that there is an
       overlay window present */
    capabilities |= OVERLAY;

}
```

If we were not able to create an overlay widget, we must revert to our alternate plan to use only the image widget and re-render the text every time the cube is re-rendered. We must destroy the overlay widget (which is really just another image planes widget) and its frame parent. Then, since the overlay widget will not exist to cover the image widget, we must add the input and button motion callbacks to the image widget. Finally, we'll set the text color and perform the other text initialization for the image widget. There is no need in this case to allocate the text color.

```
else
{
    /* Either the overlay gopen() was not successful or we could not
       get a transparent color index.  In either case, we do not want
       to use the overlay window and widget. */

    XtDestroyWidget( XtParent(sb_overlay) );

    /* Since the overlay widget will not exist to receive the user's
       input events, we'll establish those callbacks on the image widget
       instead. */

    XtAddCallback( sb, XmNinputCallback,  callback, CLICK_EVENT );
    XtAddEventHandler( sb, Button1MotionMask, FALSE, callback,
                      MOTION_EVENT );

    /* Since the text will now have to be rendered into the image
       widget instead of the overlay, we'll set up the Starbase text
       characteristics for the image widget. */

    text_color( fildes, 1.0, 0.0, 0.0 );
    initialize_starbase_text( fildes );

}
```

Now, we will establish an X context much as we did in our original example. In this case, however, the context will also contain the file descriptor for the overlay widget. We will save the context on the image planes window and on the overlay planes window (if it was indeed an overlay window).

```
/* We will initialize the contents of a context structure and save the
   context on the Starbase window so that this information can later
   be obtained when callbacks occur. */

context_data.fildes = fildes;
context_data.overlay_fildes = overlay_fildes;
context_data.capabilities = capabilities;
context = XUniqueContext();
XSaveContext( XtDisplay(sb), XtWindow(sb), context, &context_data);
if ( capabilities & OVERLAY )
    XSaveContext( XtDisplay(sb), XtWindow(sb_overlay),
                 context, &context_data);
```

Allocating an Overlay Color

If we specify that the overlay widget not use an `INIT` when it performs the Starbase `gopen`, then our overlay widget will use the default X colormap. This has the advantage of avoiding the "color flashing" effect which can be caused when clients create their own colormaps. However, it also requires our application to deliberately allocate all colors which it wishes to use in the overlay widget. If other X clients (the window manager, etc.) have allocated lots of colors of their own in the default colormap it may not be possible for our application to allocate one or more of its desired colors. Our application needs to be prepared to use alternate colors (perhaps only black and white) if necessary.

When we created our `main` function in the previous section, we called a function of our own called `allocate_overlay_color` to allocate a color for Starbase text. This `allocate_overlay_color` function will attempt to allocate a requested color (specified by the floating point `r, g, b` function parameters) and return the color index. If it is unsuccessful, it will return the index for the X color "white" instead. Refer to the *Programming with Xlib* manual for detailed information about color allocation and X colormaps.

```
/*

  allocate_overlay_color - this function is needed to allocate a color for
  the overlay widget.  Since we're using the regular X colormap, allocation
  is necessary to avoid having another X client change the color we've
  decided to use in Starbase.  The function returns the colormap index
  of the color which comes closest to the r, g, b color requested.

*/
int allocate_overlay_color( wdg, r, g, b )
    Widget wdg;
    float r, g, b;
{
    Display *display;
    Colormap colormap;
    XColor x_color;
    Status status;
    Arg arg[1];
    int colormap_index;

    /* Fetch the widget X colormap */
    XtSetArg( arg[0], XmNcolormap, &colormap );
    XtGetValues( wdg, arg, 1);

    /* Attempt to allocate the color in the X colormap */
    x_color.red   = 65535 * r;
    x_color.green = 65535 * g;
    x_color.blue  = 65535 * b;
    status = XAllocColor( XtDisplay(wdg), colormap, &x_color );

    if ( status == 0 )
    {
        /* We could not allocate the color.  Use white instead. */
        colormap_index = WhitePixelOfScreen( XtScreen(wdg) );
    }
    else
    {
        /* We did successfully allocate the color. */
        colormap_index = x_color.pixel;
    }

    return( colormap_index );

} /* end of allocate_overlay_color() */
```

Initializing the Starbase Text Characteristics

In our `main` function we called another function of our own called `initialize_starbase_text`. This simple function makes several Starbase calls to establish the appearance of text which our application will later draw.

```
/*

  initialize_starbase_text - this function establishes the characteristics
  of text rendering.

*/
void initialize_starbase_text( fildes )
    int fildes;
{
    /* Make the text centered on position */
    text_alignment( fildes, TA_CENTER, TA_HALF, 0.0, 0.0 );

    /* Make the size of the text equal to 1/20 of the overall window size */
    character_width( fildes, 0.05 );
    character_height( fildes, 0.05 );

} /* end initialize_starbase_text() */
```

Handling the Callback

We need to declare in our `callback` function a new variable, type `Boolean`, called `is_overlay`. In the `callback` function, after we have obtained the X context, we must get the overlay resource from the widget which experienced the callback. If the resource has a value of "True", then the callback is associated with the overlay widget, otherwise it is associated with the image widget.

```
    /* Determine if this is the overlay widget */

    XtSetArg( arg[0], XgNoverlay, &is_overlay );
    XtGetValues( wdg, arg, 1 );
```

Next, we need to handle the various possible callback types. Just as we did in our original example, we will use a `switch` statement with cases for the various callback types. First, we will deal with the `EXPOSE_EVENT` type. In this case, if the `is_overlay` variable is True, we'll call our own `draw_text` function to re-draw the text in the overlay planes. We'll draw a text string which says "Overlay Text". Otherwise, we need to perform a Starbase `clear_view_surface` and then call our `draw_cube` function to re-draw the 3D cube object.

Our `draw_cube` function will check for the `OVERLAY` flag in the capabilities flag word. If the flag is not set, then no overlay widget was created and our `draw_cube` function will emulate that overlay widget functionality by calling `draw_text` to render the text into the image planes widget *after* it has drawn the cube.

```
case EXPOSE_EVENT:
    if ( is_overlay )
    {
        /* This is an expose of the overlay widget.  We need to
           only redraw the text.  The window clear will already have
           been done by the X toolkit. */
        draw_text( context_data->overlay_fildes, "Overlay Text" );
    }
    else
    {
        /* This is an expose of the image widget.  We need to clear
           the window and z-buffer and re-draw the cube.  We will
           automatically call draw_text() inside the draw_cube()
           function if there is not an overlay window present. */
        clear_view_surface( context_data->fildes );
        draw_cube( context_data->fildes, context_data->capabilities );
    }
    break;
```

Next, we'll handle the `RESIZE_EVENT` case. The Starbase widget will take care of most of the work which needs to be done to account for the resize event. However, if this is an image widget resize event we need to restore the Starbase viewing state and, if our graphics device is `SOLIDS_CAPABLE`, we must restore the `hidden_surface` removal because the Starbase widget will have turned it off. Since the overlay widget uses the default viewing state and does not use hidden surface removal no extra action is required for it.

```
case RESIZE_EVENT:
    if ( !is_overlay )
    {
        initialize_camera( context_data->fildes );
        if (context_data->capabilities & SOLIDS_CAPABLE)
        {
            /* need to re-establish hidden surface removal for the
               image widget */
            hidden_surface( context_data->fildes, TRUE, TRUE );
        }
    }
    break;
```

The `CLICK_EVENT` and `MOTION_EVENT` cases are unchanged in the `callback` function. If an overlay widget exists, the text does not need to be re-drawn when the cube is rotated and re-drawn. However, if an overlay widget does not exist, the `draw_cube` function will render the text into the image planes widget *after* it has drawn the cube.

Drawing the Overlay Text

If an overlay widget was created, our `draw_text` function will be called only when expose events occur on the overlay widget. However, if an overlay widget was not created, we want our application to emulate the overlay text functionality by drawing the text into the image widget instead. In our `draw_cube` function, after rendering the 3D cube, we will check for the `OVERLAY` flag in the capabilities flag word. If the flag is not set then we will call our `draw_text` function to draw a text string which says "Image Text". Notice that this must be done *before* the double buffers are switched.

```
    if ( ! (capabilities & OVERLAY) )
    {
        /* There is no overlay widget so we must re-draw the text
           into the image widget */
        draw_text( fildes, "Image Text" );
    }
```

Now, we will create the draw_text function itself. Notice that a Starbase `flush_buffer` call is made after the `text3d` call. This is necessary to force Starbase to render the text right away. Otherwise, there may be a delay before the text is rendered. Since we are not interactively manipulating the text string, it is not necessary to use a Starbase `make_picture_current` call at this point.

```
/*

  draw_text - this function will draw the text string passed as a
  parameter and then flush the Starbase buffer to make the string
  appear.

*/
void draw_text( fildes, text )
    int fildes;
    char *text;
{

    text3d( fildes, 0.5, 0.5, 0.1, text, VDC_TEXT, FALSE);
    flush_buffer( fildes );

} /* end of draw_text() */
```

Section Four: Device-Specific Features

The following section discusses what minor differences there are between Starbase graphics devices. It is best that you first consult the chapter for your device driver in this manual. Once familiar with your Starbase graphics device driver, use the following device-specific features to help assure portability across the family of devices.

Porting from CRX to CRX-24

This section discusses the `hpgcrx` driver which supports either unaccelerated devices (CRX, Dual CRX, CRX-24 or the HP 710) or an accelerated device (CRX-24Z). If you need to select a specific mode, see the `gopen` man page in the *Starbase Reference Manual* for information on how to select either an unaccelerated or accelerated driver.

`CMAP_FULL` Mode

When rendering in `CMAP_FULL` mode on a CRX, the 8 planes can be used in either 3:3:2 or in 6|6|6 mode, depending on whether the `SB_X_SHARED_CMAP` environment variable was set or not. In 3:3:2 mode, the 8 planes are divided into three planes of red, three planes of green, and two planes of blue. In 6|6|6 mode, the colormap is divided into 40 colors for the window system and then a ramp of 216 colors with six shades each of red, green and blue. On a CRX-24 and CRX-24Z, there are 8 planes for each of the three colors. These differences should not effect your code unless your application needs to perform block operations. Refer to the "Block Operations" in the "Other Portable Techniques" section for details. The number of colors available will effect 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 CRX-24 (3:3:2 style only), the performance will be lower than when using all 24 planes. For performance and image quality, we recommend using 24 planes.

Note CRX-24Z performance does not change in `CMAP_FULL` mode depending on the visual depth (8- or 24-bit).

Colormap Sharing

The CRX has one hardware colormap. When different windows have different logical colormaps, colormap thrashing can occur when moving colormap focus (usually the cursor) from one window to another. To minimize this problem the CRX has a colormap sharing mechanism which is turned on by the `SB_X_SHARED_CMAP` environment variable. See the section on "Colormap Sharing Starbase and X Windows" in the *CRX Family of Device Drivers* chapter of this manual.

CRX-24 and CRX-24Z do not use this mechanism because they have five separate hardware colormaps (one for overlay, four for image planes) to minimize colormap thrashing. Three of the four image colormaps are used for `CMAP_NORMAL` and `CMAP_MONOTONIC` modes, and one is used for `CMAP_FULL` mode. Only when you run out of hardware colormaps will you see thrashing, when two applications with different logical colormaps have to share the same hardware colormap.

Starbase Echos

Echos on the CRX are rendered to the image bank not currently being used for image rendering. Echos on CRX-24 and CRX-24Z are rendered to the overlay planes. This may sometimes result in a slightly different visual behavior of echos, such as the color being different. Under most circumstances you will have no problem. For details on device-dependent X Windows information, see the chapter on your device in the *Starbase Device Drivers* manual.

On the CRX-24Z device, a common operation may be to have a window in the image planes with a transparent child window in the overlay planes. In this case, the Starbase entry point `inquire_capabilities` should be used. If the `IC_TRANS_WIN_IMAGE_CURSOR` bit is set, put the echo in the image plane window. If this bit is not set, put the echo in the overlay plane child window. (The bit will *not* be set for the CRX-24 device, but may be on other devices.)

Number of Color Planes

The CRX has 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 CRX-24 has 24 image planes and 8 overlay planes. The 24 image planes on the CRX-24 are organized as three banks of 8 planes each. If an 8-bit window is opened in the image planes, then the CRX-24 planes are accessed just like the CRX. If a 24-bit 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 24-bit 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 CRX-24 to CRX-24Z

The CRX-24Z is an optional accelerator for the CRX-24 device. The CRX-24 and CRX-24Z both use the `hpgcrx` device driver.

The CRX-24Z accelerated Starbase graphics device is highly compatible with the CRX and CRX-24 graphics device. This can allow applications written and delivered for CRX-24 to use the CRX-24Z accelerator without requiring different executable code. To achieve this compatibility, applications must be linked using shared libraries. Otherwise, the application can be relinked with the `hpgcrx` device driver that is installed with the PowerShade software to support CRX-24 or CRX-24Z.

Note The CRX-24 (with PowerShade) and the CRX-24Z provide optimal 3D shaded polygon performance. For a comprehensive listing of features which can be used for optimized performance, see the `CRX Family of Device Drivers` chapter in this manual.

Source Incompatibilities

The gescapes available for CRX-24Z are a superset of those available on CRX and CRX-24 with one exception. CRX-24 supports a `fill_pattern` of up to 16x16 pixels and because of hardware limitations, CRX-24Z only supports up to 4x4 pixels.

Possible behavioral differences between the CRX and CRX-24 and CRX-24Z are mostly because of hardware differences. These behavioral differences should not effect 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 CRX and CRX-24, but not by the CRX-24Z. Therefore, applications in the X environment should capture and act on expose events and redraw the image when one is received. Refer to the section "Sample Application" at the beginning of this chapter for examples of how an application can do this. X events are documented in the *Programming with Xlib* manual.

Save under is a X11 feature 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 for CRX-24Z acceleration. This feature is supported for the CRX and CRX-24.

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 CRX-24Z accelerator, there may be minor visual differences between accelerated and unaccelerated images. These minor differences are listed below. For specific information on the differences, consult the `CRX Family of Device Drivers` chapter of this manual.

- 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 and may change in future releases of the Starbase graphics library. For specific information on these differences, refer to the `CRX Family of Device Drivers` chapter of this manual.

Porting from CRX-24Z to CRX-48Z

This section discusses the `hpcrx48z` driver which supports the CRX-48Z device. The CRX-48Z is a graphics accelerator for the Series 700 workstations. The CRX-48Z is highly compatible with the CRX-24Z accelerated Starbase graphics device.

Colormap Sharing

Like the CRX-24Z, the CRX-48Z has five separate hardware colormaps (one for overlay, four for image planes). However, on the CRX-48Z the four image plane colormaps may be used for `CMAP_NORMAL`, `CMAP_MONOTONIC`, or `CMAP_FULL`, with no restriction. (The CRX-24Z has four image plane colormaps, but only one can be used for 12 or 24 bit `CMAP_FULL`, the other three are used for `CMAP_NORMAL` and/or `CMAP_MONOTONIC`). Colormap allocation is managed by the X Server, and on the CRX-48Z, the four colormaps are used in an optimal fashion such that colormap "thrashing" (if more than four colormaps are needed) is greatly reduced. The CRX-48Z does not use the `SB_X_SHARED_CMAP` environment variable, or the 6|6|6 colormap mode (as sometimes used by the CRX device).

Starbase Echos

Echos on CRX-24Z are rendered to the overlay planes. On the CRX-48Z device, echos are rendered in a set of planes separate from the overlay or image planes, and you may notice slight behavioral differences between echos on the two devices. Under most circumstances these differences will pose no problems. For details on device dependent echo information, see the chapter for your device in in this manual.

On devices with overlay planes, a common operation may be to have a window in the image planes with a transparent child window in the overlay planes. In this case, the Starbase entry point `inquire_capabilities` should be used. If the `IC_TRANS_WIN_IMAGE_CURSOR` bit is set, the echo should be associated with the `fildes` for the image plane window. If this bit is not set, the echo should be associated with the `fildes` for the overlay plane window. (The bit *will* be set for the CRX-48Z device, but may not be on other devices.)

Number of Color Planes

The CRX-24Z has 24 image planes and 8 overlay planes. The 24 image planes on the CRX-24Z are organized as three banks of 8 planes each. If an 8-bit window is opened in the image planes, then 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`. If a 24-bit window is opened, then the `bank_switch` function is used to select bank 0, 1, or 2 for block operations.

The CRX-48Z has 48 image planes and 8 overlay planes. The 48 image planes are organized as six banks of 8 planes each. If an 8-bit window is opened in the image planes, then `bank_switch` is used to select bank 0 or 1, just as on the CRX-24Z. If a 24-bit window is opened, then `bank_switch` is used to select bank 0, 1, 2, 3, 4, or 5. For details on how these banks are organized refer to the "hpcrx48z Device Driver" chapter in this manual.

Source Incompatibilities

There are very few source incompatibilities between the CRX-24Z and the CRX-48Z devices. CRX-24Z only supports fill_pattern up to 4x4 patterns. CRX-48Z supports fill_pattern sizes up to 4x4.

Section Five: Starbase Motif Widget Guidelines

Limitations on Starbase Usage

`set_p1_p2`

`hidden_surface`

`mapping_mode`

Using the Starbase widget places some limitations on the use of Starbase functions. In particular, the default rescale policy causes the widget to use the Starbase functions `set_p1_p2`, `mapping_mode`, and `hidden_surface`. If you will be using any rescale policy other than `XgRESCALE_NONE`, then you should not use `set_p1_p2` or `mapping_mode` directly. In addition, the resize action will turn off `hidden_surface`, so your resize callback or expose callback will need to restore `hidden_surface` if it should be enabled. The `set_p1_p2` call that the widget makes may not update all transformations and geometric attributes. If any such attributes are meant to be kept from one redraw to the next, they can be set again in a resize callback to reevaluate them with the new p1-p2 settings.

Using Starbase Display List to Refresh

The Starbase Display List can be very useful for remembering and later redrawing images when an expose callback is called. A series of Starbase calls can be recorded in a display list segment. The expose callback function can then use `refresh_segment(3G)` to play back the picture.

Appearance of Each Rescale Policy

Possible values of the `XgNrescalePolicy` resource include `XgRESCALE_NONE`, `XgRESCALE_DISTORT`, `XgRESCALE_MINOR`, and `XgRESCALE_MAJOR`.

The `XgRESCALE_NONE` policy leaves the Starbase p1-p2 limits at the original size at which the widget's window was created. No changes to p1-p2 or `mapping_mode` are made for resize actions. This policy is appropriate if the window size is not allowed to change. It may also be useful if the application implements its own rescale in response to the resize callback.

The remaining rescale policies change the Starbase p1-p2 limits and `map-ping_mode`. The widget will `gopen` Starbase with the window set to the `MaxWidth` and MaxHeight size. The default values for MaxWidth and MaxHeight are the screen width and height. The rescaling of output can only adapt to an area as large as MaxWidth and MaxHeight. The size of these limits can affect the amount of resources needed for the window. In particular, the Starbase library may allocate image buffers and Z-buffers that use memory in proportion to MaxWidth*MaxHeight.

The XgRESCALE_DISTORT policy changes the Starbase mapping_mode to distort the aspect ratio between x and y. It rescales the p1-p2 limits to the full window width and height, limited by MaxWidth and MaxHeight. This will show all of the VDC extent, and will use all of the window area. The resulting distortion may not be acceptable.

The XgRESCALE_MINOR policy changes the Starbase mapping_mode to not distort the aspect ratio between x and y. It rescales the p1-p2 limits to the full window width and height, limited by MaxWidth and MaxHeight. If the window's current aspect ratio does not match the MaxWidth/MaxHeight ratio, there will be unused parts of the window. This will show all of the VDC extent.

The XgRESCALE_MAJOR policy changes the Starbase mapping_mode to not distort the aspect ratio between x and y. It may rescale the p1-p2 limits to larger than the full window width and height. If the window is resized to a different aspect ratio than the MaxWidth/MaxHeight ratio, the p1-p2 limits will be scaled so the original aspect ratio is preserved, and the p1-p2 area exactly fits the larger of the window width or height. This may not show all of the VDC extent. It will fill all of the window.

If using the XgRESCALE_MAJOR rescale behavior, set the XgNmaxWidth and XgNmaxHeight resources of the Starbase widget to the same aspect ratio as the vdc_extent to ensure that the vdc_extent is placed against the upper left corner of the window. When using the XgRESCALE_MAJOR rescale behavior, allowing these aspect ratios to differ significantly may cause some or all of the vdc_extent to be clipped by the Starbase widget even though there is still available *white space* within the widget.

Using Dynamic Colormap Priorities

Most X servers can only install one colormap at a time. Only installed colormaps can control the appearance of a window. The Starbase widget will often have a different colormap than the other widgets in an application. When this occurs, only one of the Starbase widgets or the other widgets will look correct at one time. The application can control what widgets appear correct by changing the value of the XgNwmCmap resource.

The XgNwmCmap resource can have values of XgWM_CMAP_NONE, XgWM_CMAP_LOW_PRIORITY, and XgWM_CMAP_HIGH_PRIORITY. The XgWM_CMAP_NONE policy will not arrange for a Starbase widget to have its colormap installed. This setting is intended for applications which already have code to manipulate the WM_COLORMAP_WINDOWS property to communicate colormap needs to a window manager.

The settings XgWM_CMAP_LOW_PRIORITY and XgWM_CMAP_HIGH_PRIORITY can be used to make the Starbase widget's colormap less important or more important than the colormaps of other widgets in an application. Whenever the resource is changed from one value to another, the widget will update the priority of its colormap. The window manager will then reconsider which colormap or colormaps to install.

One possible behavior for an application is to make a Starbase widget's colormap more important when the pointer is inside that widget. The program can watch for pointer motion into and out of a widget by using the XtAddEventHandler function to request a callback for EnterWindowMask and LeaveWindowMask. The callback functions can then update the XgNwmCmap resource value. The following example program does this.

```
#include <stdio.h>
#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Frame.h>
#include <Xm/DrawingA.h>
#include <Xg/Starbase.h>

void quit(widget, client_data, call_data)
Widget    widget;
caddr_t   client_data;
caddr_t   call_data;
{
    exit(0);
}

void expose(widget, client_data, call_data)
Widget    widget;
caddr_t   client_data;
XmDrawingAreaCallbackStruct *call_data;
{
    Arg arg[10];
    int n;
    int fildes;

    /* Find the Starbase file descriptor. */
    n = 0;
    XtSetArg(arg[n], XgNfildes, &fildes); n++;
    XtGetValues(widget, arg, n);

    if (fildes < 0)
        return;

    clear_view_surface(fildes);
    fill_color(fildes, 0.3, 0.5, 0.2);
    ellipse(fildes, 0.3, 0.4, 0.5, 0.5, 0.7);
    flush_buffer(fildes);
}

void enter(widget, client_data, event)
Widget widget;
caddr_t client_data;
XEvent *event;
{
    Arg arg[10];
    int n;

    n = 0;
    XtSetArg(arg[n], XgNwmCmap, XgWM_CMAP_HIGH_PRIORITY); n++;
    XtSetValues(widget, arg, n);
}


void leave(widget, client_data, event)
Widget widget;
caddr_t client_data;
XEvent *event;
{
    Arg arg[10];
    int n;

    n = 0;
    XtSetArg(arg[n], XgNwmCmap, XgWM_CMAP_LOW_PRIORITY); n++;
    XtSetValues(widget, arg, n);
}


main(argc, argv)
int argc;
char *argv[];
{
    XtAppContext app_context; /* application context */
    Widget toplevel, outer, quit_button, frame, sb;
    Arg arg[15];
    int n;

    toplevel = XtAppInitialize(&app_context, "DRawingArea", NULL, 0,
        &argc, argv, NULL, NULL, 0);

    n = 0;
    outer = XtCreateManagedWidget(NULL, xmFormWidgetClass, toplevel, arg, n);

    n = 0;
    XtSetArg(arg[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNlabelString,
        XmStringCreate("Quit", XmSTRING_DEFAULT_CHARSET)); n++;
    quit_button = XtCreateManagedWidget("quit", xmPushButtonWidgetClass,
        outer, arg, n);
    XtAddCallback(quit_button, XmNactivateCallback, quit, NULL);

    n = 0;
    XtSetArg(arg[n], XmNtopWidget, quit_button); n++;
    XtSetArg(arg[n], XmNtopAttachment, XmATTACH_WIDGET); n++;
    XtSetArg(arg[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(arg[n], XmNshadowThickness, 5); n++;
    frame = XtCreateManagedWidget("frame", xmFrameWidgetClass,
        outer, arg, n);

    n = 0;
    XtSetArg(arg[n], XgNshadeMode, XgCMAP_MONOTONIC); n++;
    sb = XtCreateManagedWidget("Starbase", xgStarbaseWidgetClass,
        frame, arg, n);
    XtAddCallback(sb, XmNexposeCallback, expose, NULL);
    XtAddEventHandler(sb, EnterWindowMask, False, enter, NULL);
    XtAddEventHandler(sb, LeaveWindowMask, False, leave, NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_context);
}
```

3 HP VMX Device Driver

Introduction

The `hpvmx` Starbase driver, or HP VMX, offers application developers and end users an exciting and powerful new 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.

The section "Device Description" will provide you with information to help you answer questions like:

- What is HP VMX?
- How do you use HP VMX?
- How does HP VMX work?

With this information, you will be able to determine how to utilize the capabilities of HP VMX. The remaining sections in this chapter provide additional details on the usage of HP VMX.

Note HP VMX is supported on 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.

HP VMX/PowerShade Licensing

To take advantage of the remote capabilities of HP VMX *and* PowerShade, you must purchase "PowerShade for the HP700/RX X Station." This license allows you to run PowerShade applications to any remote X11 client; however, this license is *not* required to run HP VMX on your local display.

PowerShade

The 3D surfaces software, PowerShade (B2156C), is fully supported on HP VMX. By combining PowerShade and 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
- Double buffering (8 planes per buffer)

And you can do this all within 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 8-bit 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)
- HP Series 300/400 Workstations running X11

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:

- *Starbase Graphics Techniques* — explains backing store in X Windows.
- *Programming on HP-UX* — covers linking shared or archive libraries.

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 *X11 Windows* 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. HP VMX accomplishes this in two steps:

1. HP VMX renders graphics images into virtual memory.
2. HP VMX displays these images in the targeted window using standard X11 protocol.

Because HP VMX uses the X11 protocol to display the images, this targeted window may be local or remote on: HP or non-HP hardware, a workstation, an X terminal, or a PC[2]. The only requirement is that you run 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 "HP VMX vs. SOX11" 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 VMX. This example is intended to give you a feel for the kinds of steps necessary to use VMX, rather than provide a detailed tutorial on all the steps necessary to explain each step. Refer to the sections throughout this chapter, including, “HP VMX Licensing”, “To Compile and Link with the Device Driver”, and “To Open and Initialize the Device for Output” for details on these steps.

HP VMX Usage Example

In order 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 permission to access your X Station's local display server. For example,

   ```
   xhost +hpdspsvr
   ```
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. For example, in `ksh`:

   ```
   export DISPLAY=hpxterm:0.0
   ```
6. Run your application from this window.

The application is now executing on the HP 735 (`hpdspsvr`), and displaying X and Starbase output on the X Station (`hpxterm:0.0`).

This example illustrates that it is easy to run your Starbase applications across the network in the X11 client-server model using HP VMX.

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 render these Starbase 3D graphics images to a virtual memory frame buffer and display these images to an X11 window using standard X11 protocol.

Here are the basic steps HP VMX performs:

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 supports only 8/8 double buffering).

HP VMX Configurations

The HP VMX driver supports only 8-bit X11 windows. Attempts to `gopen` windows with a depth other than 8 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

HP VMX Device Driver, VM Rendering Utilities, and Overlay Planes

Before going further, we must clarify some points related to HP VMX. In order to do so, you need to understand the two basic functions that HP VMX provides:

- To render Starbase graphics into a virtual memory frame buffer.
- To then display this VM frame buffer in the targeted X11 window.

Together, these two functions create what we call "HP VMX".

VM Rendering Utilities

There also 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 (Virtual Memory) 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 "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 CRX Family Overlay Plane Driver" for details on how HP VMX is used in this capacity.

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. The HP VMX driver is *not* supported in raw mode.

To Compile and Link with the Device Driver

For Shared Libraries

The compiler driver programs (`cc`, `f77`, `pc`) link with shared libraries by default. The HP VMX shared library device driver is the file named `libddvmx.sl` in the `/usr/lib` directory. Starbase will explicitly load the device driver at run time when you compile and link with the Starbase shared library `/usr/lib/libsb.sl`, or use the `-lsb` option. This loading occurs at `gopen(3G)` time.

Since HP VMX is supported only for X Windows, the window libraries must be linked in with all programs that use HP VMX.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/X11\
   -L/usr/lib/X11R5 -lXwindow -lsb\
   -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN 77 use,

```
f77 example.f -Wl,L/usr/lib/X11R5 -lXwindow -lsb\
    -lXhp11 -lX11 -ldld -lm -o example
```

or with Pascal use,

```
pc example.p -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
      -lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the gopen procedure in the section "To Open and Initialize the Device for Output" in this chapter.

3

For Archive Libraries

The HP VMX archive library device driver is part of the Starbase archive library /usr/lib/libsb1.a. Including the HP VMX archive library device driver in libsb1.a ensures that all archive library applications have access to the HP VMX capabilities without explicitly including the HP VMX device driver in the link sequence.

Note that the implementation of the VM Rendering Utilities, and the use of VMX as the overlay plane device driver on the CRX family of devices necessitates the existence of HP VMX within the libsb1.a.

By default, the linker program ld(1) looks for a shared library driver first and then the archive driver if a shared library was not found. By exporting the LDOPTS variable, the -l option will refer only to archive drivers.

Examples

Assuming you are using ksh(1), to compile and link a program for use with HP VMX and the hpgcrx device driver, use:

```
export LDOPTS='-a archive'
```

and then:

```
cc example.c -lddgcrx\
   -L/usr/lib/X11R5 -lXwindow\
   -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

or for FORTRAN 77, use:

```
f77 example.f -lddgcrx\
    -Wl,-L/usr/lib/X11R5 -lXwindow\
    -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

or for Pascal, use:

```
pc example.p -lddgcrx\
   -Wl,-L/usr/lib/X11R5 -lXwindow\
   -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

Again, note there is *no* reference to `-lddvmx` needed since the HP VMX code resides in `/usr/lib/libsb1.a`.

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 use this X11 environment to run an application with HP VMX. 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.

xhost Command

The `xhost` command is used to add or delete a remote host's permission to access the local display server. This command must be run on the HP VMX client side to allow the HP VMX server access to the HP VMX client's display server.

Licensing

In order to run a PowerShade application on any remote X11 server, you must purchase the "PowerShade for HP700/RX X Station" product. Refer to the section "HP VMX/PowerShade Licensing" for details.

Syntax Examples

Two methods exist to `gopen` a window using HP VMX. One method for `gopening` HP VMX is to specify `hpvmx` as the *driver* parameter to `gopen()`. The second method is to set the *driver* parameter to `NULL` and let Starbase choose the appropriate device driver. 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 family of devices (for example, CRX-24, CRX-24Z, CRX-48Z)

In each of the examples below, assume that the window `/dev/screen/remote_window` has been created on a remote X11 server with the following `xwcreate` command:

```
xwcreate -display <remote_host> -geometry 500x500 remote_window
```

C programs

```
fildes = gopen("/dev/screen/remote_window", OUTDEV, NULL, INIT);
```

FORTRAN 77 programs

```
fildes = gopen('/dev/screen/remote_window'//char(0), OUTDEV,
               char(0), INIT)
```

Pascal programs

```
fildes := gopen('/dev/screen/remote_window', OUTDEV, '', INIT);
```

Parameters for gopen

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

path — 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 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 a NULL parameter. In this case, Starbase will automatically choose the appropriate driver.

For example,

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

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 action is taken: 1. The software color table is initialized from the X colormap 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 0s. 2. A new X colormap is created and associated with this window. The colormap 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.

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

Dither Default

The number of colors allowed in a dither cell is 1, 2, 4, 8 or 16. The default value is 16. Selecting a color with the `fill_color` procedure will allow dithering for filled areas when desired.

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 3-1. Line Type Defaults

Line Type	Pattern
0	1111111111111111
1	1111111100000000
2	1010101010101010
3	1111111111111010
4	1111111111101010
5	1111111111100000
6	1111111111110110
7	1111111110110110

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 3-2. 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 no-ops).

```
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 using PowerShade with HP VMX, a much wider set of functionality is supported. The following calls are not supported when using PowerShade with HP VMX:

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

Conditional Support of Starbase Calls

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 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 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 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.

`double_buffer`	HP VMX supports only 8/8 double-buffering. Attempts to double-buffer with less than 8 planes will default to 8/8 double-buffering.
`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 colormap 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, the *use* parameter must be set to zero. Any extra coordinates will be ignored. If using PowerShade software, `vertex_format` is fully functional.
`with_data`	routines

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

Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored since the device does not support contouring.

Supported Gescapes

The hpvmx driver supports the following gescape operations. Refer to Appendix A of this 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 ops.
R_BIT_MODE	Enable/disable bit mode block ops.
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.
R_LOCK_DEVICE	Lock device.
R_UNLOCK_DEVICE	Unlock device.
READ_COLOR_MAP	Copy the hardware colormap into the software color map.
SWITCH_SEMAPHORE	Semaphore control.
TRIGGER_ON RELEASE	Trigger only when button is released.

Additional Gescapes Supported with PowerShade

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)
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.
SWITCH_SEMAPHORE, R_LOCK_DEVICE, R_UNLOCK_DEVICE	Because HP VMX renders to a virtual memory frame buffer, device locking is not necessary. These gescapes, therefore 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:

- `make_picture_current`
- `flush_buffer`
- `dbuffer_switch` (if double-buffering is enabled)

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. The performance considerations for those drivers may not apply to HP VMX.

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 8-bit X11 windows, the frame buffer is allocated on a byte-per-pixel basis.

For example, consider an X11 graphics window which is 750 pixels wide and 600 pixels high. The size of the VM frame buffer is:

750 pixels * 600 pixels * 1 byte/pixel = 450,000 pixels

450,000 pixels * 1 byte/pixel = 450,000 bytes

So the VM frame buffer for this window consumes .45 Mbytes of virtual memory.

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:

- Many windows (especially large windows) are `gopen`'d simultaneously.
- Your system has a small amount of RAM.

In order to alleviate these problems, you may:

- Use the hardware device driver whenever possible. For example, if you are running on a CRX system, specify `hpgcrx` 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 on the CRX family of devices. (If your *driver* parameter equals `NULL`, this happens automatically.)
- Reduce the number of simultaneous `gopen`s (`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.

Restricted gopens

As with VM (Virtual Memory) double-buffering, multiple `gopen`s of HP VMX to the same window should not be attempted. The VM frame buffer allocated by HP VMX is associated with each `gopen` rather than with a window. Multiple `gopen`s, therefore, 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

It is important to note that this section covers VM Rendering Utilities and *not* HP VMX driver functionality.

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 exercise 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 utilities 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 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 is this mode uses four planes at a time. Also, X11 does not support 4/4 double-buffering, so where your graphics window double-buffers as expected, the rest of your windows flash. Note that 4/4 double-buffering is not supported in `CMAP_FULL`.

8/8 double-buffering enhanced performance

Virtual memory (VM) 8/8 double-buffering is supported by setting the `SB_710_VM_DB` environment variable to `TRUE`. (Note that the environment variable `SB_710_VM_DB` applies to the Models 705, 715 and 725 as well). 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 window flashing 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 algorithms. As a result, rendering to the VM buffer is somewhat slower for many operations and significantly slower for a few operations such as rendering 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. 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).
3. Restricted `gopens` - Multiple output `gopens` to the same window should not be attempted. This is because the VM buffer is associated with each `gopen` rather than with the window.

To enable VM double-buffering

There are two ways you can enable VM double-buffering on the HP Models 705, 710, 715 and 725.

- You need to define the `SB_710_VM_DB` variable in your environment *before* starting your application. For example, using **`ksh`** syntax, execute the following:

  `export SB_710_VM_DB=TRUE` *or*

- 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. 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 upon detection of a window expose event.

Note HP VMX only supports 8-plane rendering, and therefore is only used for backing store of 8-plane image windows on the CRX family.

Refer to the “Backing Store Operation” section of the *Starbase Graphics Techniques* manual for more information on backing store.

HP VMX: The CRX-family 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 *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 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 3-3 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 `hpgcrx`, `hpcrx48z`, `hpvmx`, or `NULL` driver is specified.

Table 3-3. Driver Selection at gopen

Type of Window	Driver Specified	Window Depth	Driver Used
Image	`hpgcrx` or `hpcrx48z`	8 or 24	`hpgcrx` or `hpcrx48z`
Image	`hpvmx`	8 24	`hpvmx` not supported
Image	`NULL`	8 or 24	`hpgcrx` or `hpcrx48z`
Overlay	`hpgcrx` or `hpcrx48z`	8	not supported
Overlay	`hpvmx`	8	`hpvmx`
Overlay	`NULL`	8	`hpvmx`

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

Performance differences also exist between HP VMX and SOX11. These differences stem from the differences in implementation of the two drivers. HP VMX renders to a VM frame buffer and then displays the contents of that buffer to an X11 window. SOX11, on the other hand, implements each of the supported Starbase output commands directly through Xlib. SOX11 does not use a VM frame buffer.

Wireframe performance, for example, may be better in some cases on SOX11 than with HP VMX. In this case, SOX11 has less work to do by sending Xlib calls to render a few vectors, than does HP VMX in displaying the entire VM frame buffer into which it has rendered the vectors. In general, simple, sparse images may be faster with SOX11 since less work is involved in implementing a few primitives via Xlib than displaying an entire VM frame buffer.

SOX11 performance is significantly less than HP VMX for many other operations, and as discussed above, SOX11 supports fewer Starbase features than HP VMX.

4 The CRX Family of Device Drivers

High Performance Grayscale and Color Graphics

The `hpgcrx` driver supports a family of similar devices. These devices have different levels of hardware and SPU 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.
- Filling rectangles (flat shading).
- Filling shaded polygons.
- Performing 3-D transformations.
- Clipping.
- Dithering.

The `hpgcrx` device driver supports the high performance graphics devices listed below.

hpgcrx Devices For the Series 700

The Series 700 workstations have four configurations of integrated graphics (see Table 4-1). Note that each display description given in Table 4-1 has eight planes.

Note Raw-mode graphics support will *not* be provided on any Series 700 graphics device at the HP-UX 10.0 release. CRX and GRX will support raw-mode graphics until HP-UX 10.0. If you are an application developer, you are strongly encouraged to move to X11. For information on Starbase with X11, read the chapter "Using Starbase with the X Window System" in the *Starbase Graphics Techniques* manual.

Table 4-1.
Integrated Graphics Configurations for Series 700 Workstations

Display Description	Series 9000 Model Numbers			
	705	710	715	725
1024x768 color	●	●	●	●
1280x1024 grayscale	●	●		
1280x1024 color			●	

The `hpgcrx` device driver also supports:

- The GRX (A1924A) is an 8/8-plane, 1280x1024 pixel grayscale device.
- The CRX (A1659A) is an 8/8-plane, 1280x1024 pixel color display.
- The Dual CRX (A2269A) is a dual-headed 8/8-plane, 1280x1024 pixel color display. The two CRX devices are on a single board, requiring only one hardware expansion slot.
- The CRX-24 (A1439A) is a 1280x1024 pixel, direct color display with 24 image planes and 8 overlay planes.
- The CRX-24Z is the CRX-24 (A1439A) 24-plane, direct color graphics card with an accelerator (A1454A), providing high performance 3D solids modeling and anti-aliasing.

hpgcrx Devices For the Series 400

- The GRX (A1924A) is an 8/8-plane, 1280x1024 pixel grayscale device.
- The CRX (A1659A) is an 8/8-plane, 1280x1024 pixel color display.

Other Information

The integrated grayscale and integrated color displays have a single-buffer of 8 planes. The GRX, CRX and Dual CRX have two buffers of 8 planes each for double-buffering. The CRX-24 and CRX-24Z have 24 planes single-buffered or 12/12 planes double-buffered.

In order to reduce flickering, these graphics devices refresh the attached CRT displays at 72 Hz. The medium resolution HP 710 (A2210A) refreshes the display at 75 Hz.

8/8 Double-Buffering on the Integrated Displays with PowerShade

Virtual Memory Double-Buffering

Virtual memory double-buffering information for the following topics can be found in the section "VM Double-Buffering on 8-plane Devices" in the chapter "HP VMX Device Driver" in this manual:

- 4/4 double-buffering limitations
- 8/8 double-buffering enhanced performance
- How it works
- Tradeoffs
- Enabling VM double-buffering

PowerShade

The 3D surfaces software, PowerShade (B2156C), works with all of the Series 700 color devices in the CRX family. PowerShade does not work with the CRX on the Series 400.

Table 4-2. Displays Support by the `hpgcrx` Driver

Device Name	Device File Names
HP 705 19-inch Grayscale	`/dev/screen/<`*dev_name*`>`
HP 705 19-inch Color	`/dev/screen/<`*dev_name*`>`
HP 705 16-inch Color	`/dev/screen/<`*dev_name*`>`
HP 710 19-inch Grayscale	`/dev/screen/<`*dev_name*`>`
HP 710 19-inch Color	`/dev/screen/<`*dev_name*`>`
HP 710 16-inch Color	`/dev/screen/<`*dev_name*`>`
HP 715 19-inch Grayscale	`/dev/screen/<`*dev_name*`>`
HP 715 19-inch Color	`/dev/screen/<`*dev_name*`>`
HP 715 16-inch Color	`/dev/screen/<`*dev_name*`>`
HP 725 19-inch Gray	`/dev/screen/<`*dev_name*`>`
HP 725 19-inch Color	`/dev/screen/<`*dev_name*`>`
HP 725 16-inch Color	`/dev/screen/<`*dev_name*`>`
GRX	`/dev/screen/<`*dev_name*`>`[1]
CRX	`/dev/screen/<`*dev_name*`>`[1]
Dual CRX	`/dev/screen/<`*dev_name*`>` `/dev/screen/<`*dev_name*`>`
CRX-24	`/dev/screen/<`*dev_name*`>`
CRX-24Z	`/dev/screen/<`*dev_name*`>`

1 CRX and GRX will be supported in raw mode (using device path `/dev/crt`) until HP-UX 10.0.

For More Information

Information in this chapter is device-specific. For more detailed information on the areas listed below, please refer to the noted documents:

- See the *Starbase Graphics Techniques* manual to read about backing store in X Windows.
- Refer to the *Programming on HP-UX* manual to read about linking shared or archive libraries.
- See the *Fast Alpha/Font Manager Programmer's Manual* to read about how this device driver supports raster text calls from the Fast Alpha and Font Manager libraries.

Device Descriptions

HP Series 700 Integrated Graphics

The HP Series 700 Models 705, 710, 715 and 725 workstations support three different versions of integrated graphics.

High Resolution Grayscale

These displays have 1280x1024 pixels with a single bank of eight planes for 256 shades of gray. There is no offscreen memory in the frame buffer. They are 8-plane single-buffered devices. These grayscale devices are similar to the grayscale GRX, but there is not an extra 8-plane bank for double-buffering.

The only X server mode supported is image mode with 8 planes, single-buffered. No special settings are needed in the *X0screens* file.

High Resolution Color

This display has 1280x1024 pixels with a single bank of 8 color planes for 256 colors. It has a color map that provides 8 bits per color (for red, green and blue components). This yields a color palette of over 16 million colors. There is no offscreen memory in the frame buffer. The HP 705/710/715/725 color are 8-plane single-buffered devices. These color models are very similar to the CRX, but there is not an extra 8-plane bank for double-buffering.

The only X server mode is image mode with 8 planes, single-buffered. No special settings are needed in the *X0screens* file.

Medium Resolution Color

This display has a resolution of 1024x768 pixels. In all other respects, it is the same as the HP 710 color display (A2213A).

Grayscale GRX (A1924A)

This display has 1280x1024 pixels with two banks of eight planes. Its color map provides eight bits for a total of 256 values of gray. There is no offscreen memory in the frame buffer. This device is supported on both the Series 400[1] and the Series 700 workstations.

The GRX device is an 8-plane, double-buffered grayscale version of the CRX. It provides the same level of performance as the color CRX in a grayscale configuration; the GRX only supports a grayscale monitor.

The software to handle grayscale actually resides in the higher level Starbase code, not the driver. It is designed so that color applications can run without modification using the closest matching shades of gray. Application optimization for grayscale is encouraged, but not necessary. All the Starbase color map modes (`CMAP_NORMAL`, `CMAP_FULL` and `CMAP_MONOTONIC`) are supported on the grayscale configurations.

The only X server mode supported is image mode with eight planes, double-buffered. No special settings are needed in the *X0screens* file.

Starbase echos are stored in the opposite bank from the bank being rendered to. This can cause some anomalies in echo appearance. X runs in one bank and Starbase single-buffered applications run in the other bank.

If the GRX is used primarily for Starbase graphics applications in X, it is recommended that the `SB_X_SHARED_CMAP` environment variable is set to true. (See the *Color Map Sharing Starbase and X Windows* section). If the primary usage of the GRX is X applications that do not use Starbase, then do not set the `SB_X_SHARED_CMAP` environment variable.

[1] The Series 400t requires an SGC adapter and the Series 400s should be ordered with the integrated CRX/GRX option or with the SGC connector option.

Color CRX (A1659A)

This display has 1280x1024 pixels with two banks of eight color planes each for 256 colors. Its color map provides eight bits per color (for red, green and blue components). This yields a color palette of over 16 million colors. There is no offscreen memory in the frame buffer. This device is supported on both the Series 400[2] and the Series 700 workstations.

The CRX device supports 8/8 double-buffering. It is similar to the PersonalVRX (HP 98705B) except that the CRX does not have overlay planes, and each window can display a bank independently of other windows. Another term for this second feature is *per-window double-buffering*.

The CRX device has some differences due to its color map hardware support in `CMAP_FULL` mode. See the section on color in this chapter.

The only X server mode supported is image mode with eight planes, double-buffered. No special settings are needed in the *X0screens* file.

Starbase echos are stored in the opposite bank from the bank being rendered to. This can cause some anomalies in echo appearance. X runs in one bank and Starbase single-buffered applications run in the other bank.

If the CRX is used primarily for Starbase graphics applications in X, we recommend that you set the `SB_X_SHARED_CMAP` environment variable to true. (See the *Color Map Sharing Starbase and X Windows* section). If the primary usage of the CRX is X applications that do not use Starbase, then do not set the `SB_X_SHARED_CMAP` environment variable.

[2] The Series 400t requires an SGC adapter and the Series 400s should be ordered with the integrated CRX/GRX option or with the SGC connector option.

CRX-24 (A1439A)

The CRX-24 device has 24-image planes and 8-overlay planes. The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer. This graphics display board is only supported on the Series 700 workstations.

You can configure the image planes in three basic ways:

8-bit indexed color (`CMAP_NORMAL`, `CMAP_MONOTONIC`), 8/8 double-buffered

12-bit direct color (`CMAP_FULL`), 12/12 double-buffered

24-bit direct color (`CMAP_FULL`), single-buffered

You can select each of the three configurations on a per-window basis. The configuration selected is a function of the depth of the window created and double-buffer mode.

In the 8-bit indexed configuration, 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 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 but the driver performs much slower than either 12-plane direct color or 24-plane direct color mode. The PowerShade software adds support for dithering, shading and Z-buffering.

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

In the 24-bit direct color configuration, a pixel is represented by eight bits each per color channel. Double-buffering is not possible in this mode.

The overlay planes are mainly for use by the X server, but Starbase can access them as a separate 8-plane graphics device. Using the overlay planes is similar to using the image planes in the 8-bit mode. Double-buffering and direct color (`CMAP_FULL` color map mode) are not supported in the overlay planes.

The X server works only in combined mode. No special settings are needed in the *X0screens* files.

There are four different hardware color maps available for use with the image planes. This reduces color flashing effects that occur when shifting the color map focus from window to window. This phenomenon is sometimes referred to as the *technicolor effect*. One of these four color maps will be dedicated for use by direct color graphics windows. The other three will be shared by all indexed color graphics processes.

Because the CRX-24 supports multiple color maps, there is less need for color map sharing between X and Starbase so the `SB_X_SHARED_CMAP` environment variable is ignored.

This display does not support graphics rendering to the raw device. Only rendering to an X window is supported.

CRX-24Z (A1439A and A1454A)

The CRX-24Z is an optional accelerator that attaches to the CRX-24 device to provide high performance 3D solids modeling and high performance 3D wireframe with anti-aliasing. The CRX-24Z accelerator has a dedicated 24-bit Z-buffer. The CRX-24Z board attaches to the CRX-24 device directly, and does not require a separate SGC slot. The `hpgcrx` driver automatically uses the CRX-24Z accelerator if it is present. The primary use of the CRX-24Z accelerator is for 3D solids modeling, including drawing Starbase polygons, rectangles, triangle strips, quadrilateral meshes, and spline surfaces.

The CRX-24Z device is a high-speed scan converter that accelerates rendering into the CRX-24 frame buffer. The CRX-24Z accelerator can render to the 24 image planes on the CRX-24 in 8-bit indexed (`CMAP_NORMAL`), 12-bit direct color (`CMAP_FULL`), and 24-bit direct color (`CMAP_FULL`). The CRX-24Z accelerator does *not* support rendering to the CRX-24 overlay planes. If the CRX-24 overlay planes are specified in a `gopen` call, the CRX-24Z accelerator will not be enabled.

Optimized 3D Shaded Polygon Performance

The CRX-24 (with PowerShade) and CRX-24Z provide optimal 3D shaded polygon performance. Shaded polygons on the CRX-24 and CRX-24Z are highly optimized and very fast in most cases. See `/usr/lib/starbase/perf.notes` in the on-line PowerShade performance notes for a detailed list of features and performance data. This data is subject to change without notice.

Dual CRX

The Dual CRX device has two color CRX displays on a single board which fits into a single expansion slot. Each display has 1280x1024 pixels with two banks of eight color planes for 256 colors. The Dual CRX is supported on the Series 700 only.

Each of the two devices on a Dual CRX requires its own device special file to access it. Mixing a Dual CRX display with any other display in the same SPU is not supported. Two Dual CRX displays in the same SPU for a total of four heads are supported.

The only X server mode supported is image mode with 8 planes, double-buffered. No special settings are needed in the *X0screens* file.

Each display behaves the same as a single color CRX with the following exceptions:

- Resetting the ITE on one display will reset them both. Any Starbase application which performs a reset when it is run may cause graphics, including X windows, on the other display to disappear.
- There is no device file shipped for the second display. The system is originally shipped with `/dev/crt` only. You must create the second device file using the `mknod` command.
- The `/usr/lib/X11/X0screens` file must be modified to run an X server which uses both displays. By default, the X server will only use the left device.

Running two X servers, one on each display of the Dual CRX, is not supported. A single keyboard and pointing device are associated with the X server, whether the server runs on one, two, three or all four displays.

This display does not support graphics rendering to the raw device. Only rendering to an X window is supported.

Supported Dual Head Configurations

The HP-UX 8.07 revision supports the following dual heads on the Series 700:

Model 720	One Dual CRX, provides two CRX heads.
Model 730	One Dual CRX, provides two CRX heads.
Model 750	Two CRX-24 heads, each requires a separate SGC slot.
Model 750	One or two Dual CRX, provides a two or four head CRX workstation.

PowerShade, 3D Surfaces Software

PowerShade is an optional software package that supports lighting and shading in graphics design. It has capabilities for both surface rendering and volumetric rendering, the latter on the CRX-24 only. The PowerShade software is included with the CRX-24Z bundles (not standalone), and is an option for the CRX, Dual CRX, CRX-24 and the color versions of the integrated graphics systems (HP 705/710/715/725). It is not supported on the GRX or the grayscale version of the integrated graphics systems (HP 705/710/715/725).

In order to use the HP VMX driver with PowerShade from any graphics system, you must *install* the PowerShade software. For more information on HP VMX, read the chapter "HP VMX Device Driver."

Interactions with the ITE

There may be interactions when both the ITE and Starbase are accessing the display simultaneously. On all displays, a hard ITE reset (Shift CTRL Reset) will clear all planes in bank 0 and it will also log you out of VUE (reset your X11 server). This hard reset operation will also clear up bad hardware states that may occur when you prematurely abort a graphics process.

Pressing CTRL Insert char disables and enables the planes where the ITE displays text. Pressing CTRL Delete char disables and enables the remaining planes. Figure 1-1 shows what planes are affected by this.

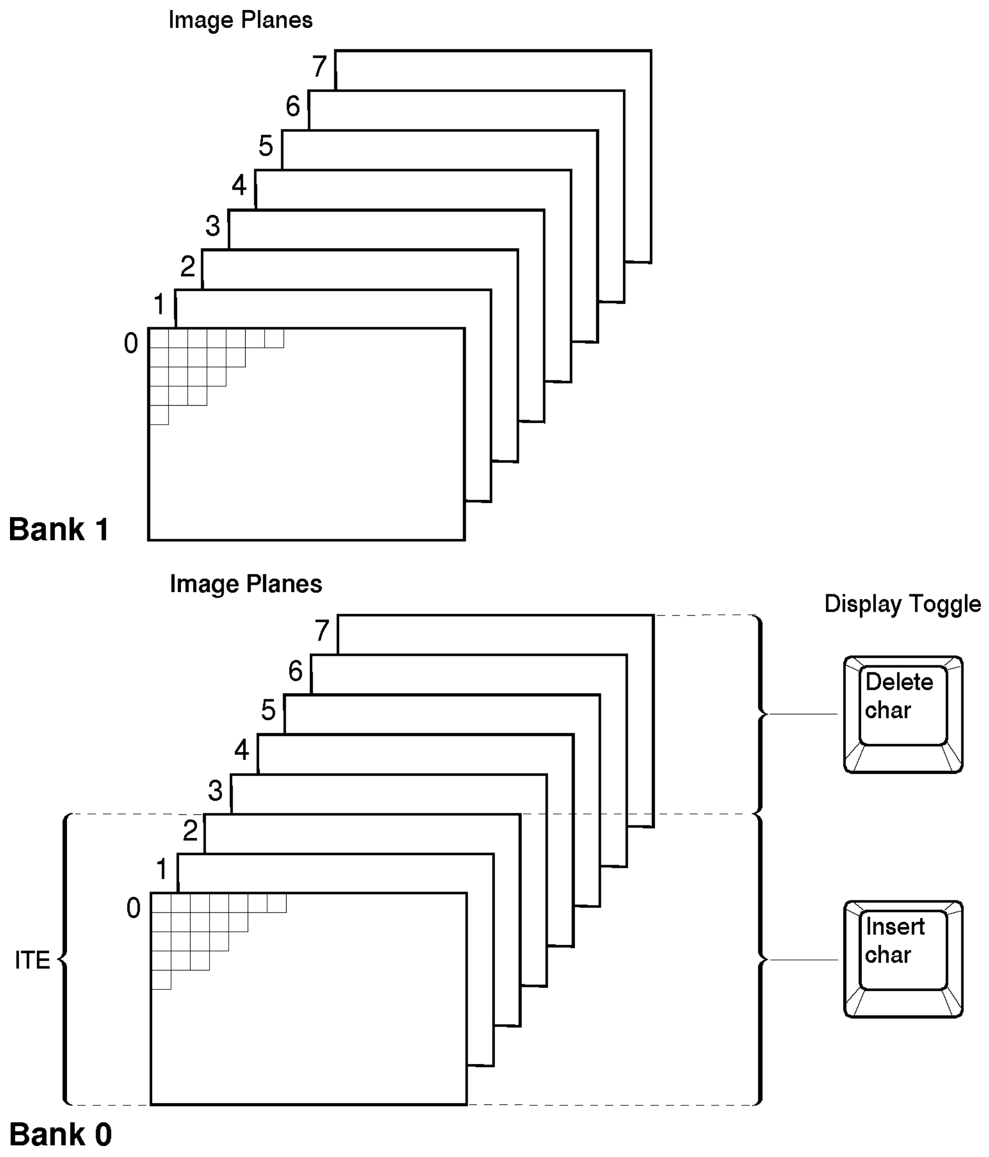

Figure 4-1.
Controlling Display (CRX, Dual CRX, GRX, HP 705, HP 710, HP 715, HP 725)

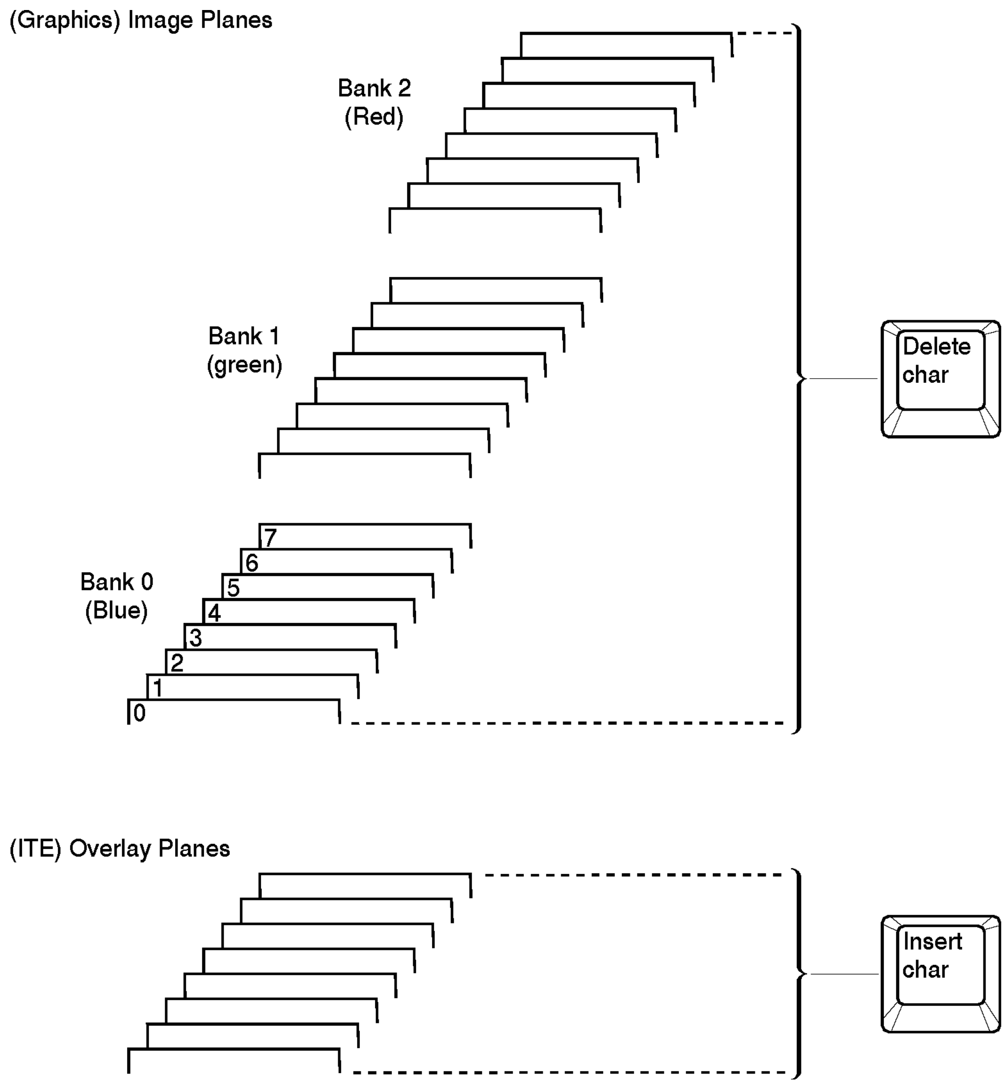

Figure 4-2. Controlling Display of the CRX-24 and CRX-24Z

Escape Sequences for Controlling the ITE

The following escape sequences cause the ITE to modify how it uses the display. Note that, in Table 4-3, even though the upper- and lower-case characters within the square brackets perform the same function they are different in that the upper-case character denotes the last command in an escape sequence.

Table 4-3. ITE Escape Sequences

Escape Sequence	Function
$^{E}{}_{C}$`&v_[p | P]`	constrain ITE to operate in **n** planes (n = 1, 2, or 3) (The default is n = 1 for monochrome displays, and n = 3 for color displays.)
$^{E}{}_{C}$`*d[a | A]`	clear graphics display
$^{E}{}_{C}$`*d[c | C]`	enable full graphics display
$^{E}{}_{C}$`*d[d | D]`	disable full graphics display
$^{E}{}_{C}$`*d[k | K]`	enable display of graphics cursor
$^{E}{}_{C}$`*d[l | L]`	disable display of graphics cursor
$^{E}{}_{C}$`*d[e | E]`	enable full alpha (ITE) display
$^{E}{}_{C}$`*d[f | F]`	disable full alpha (ITE) display
$^{E}{}_{C}$`*d[q | Q]`	enable display of alpha (ITE) cursor
$^{E}{}_{C}$`*d[r | R}`	disable display of alpha (ITE) cursor

The Frame Buffer

Physical Address Space

The physical frame buffer is addressed as 2048x1024 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.

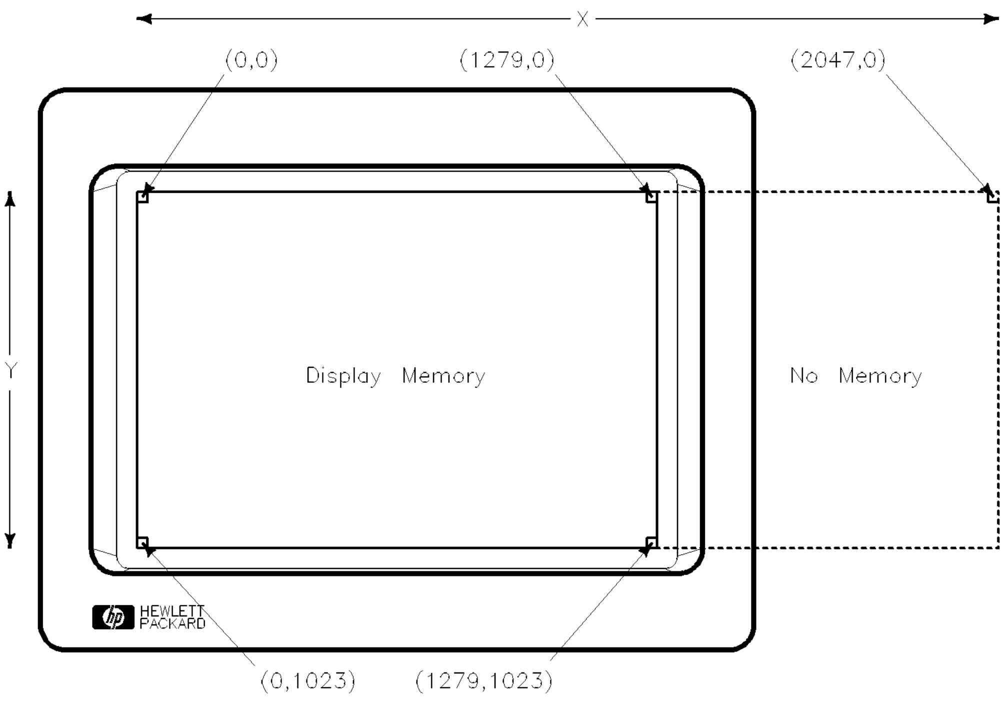

Figure 4-3. Physical Address Space

The HP 705, HP 710, HP 715 and HP 725 are addressed the same as above except that the display memory is only 1024x768 pixels.

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 unexpected 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 (1023x2048+1279).

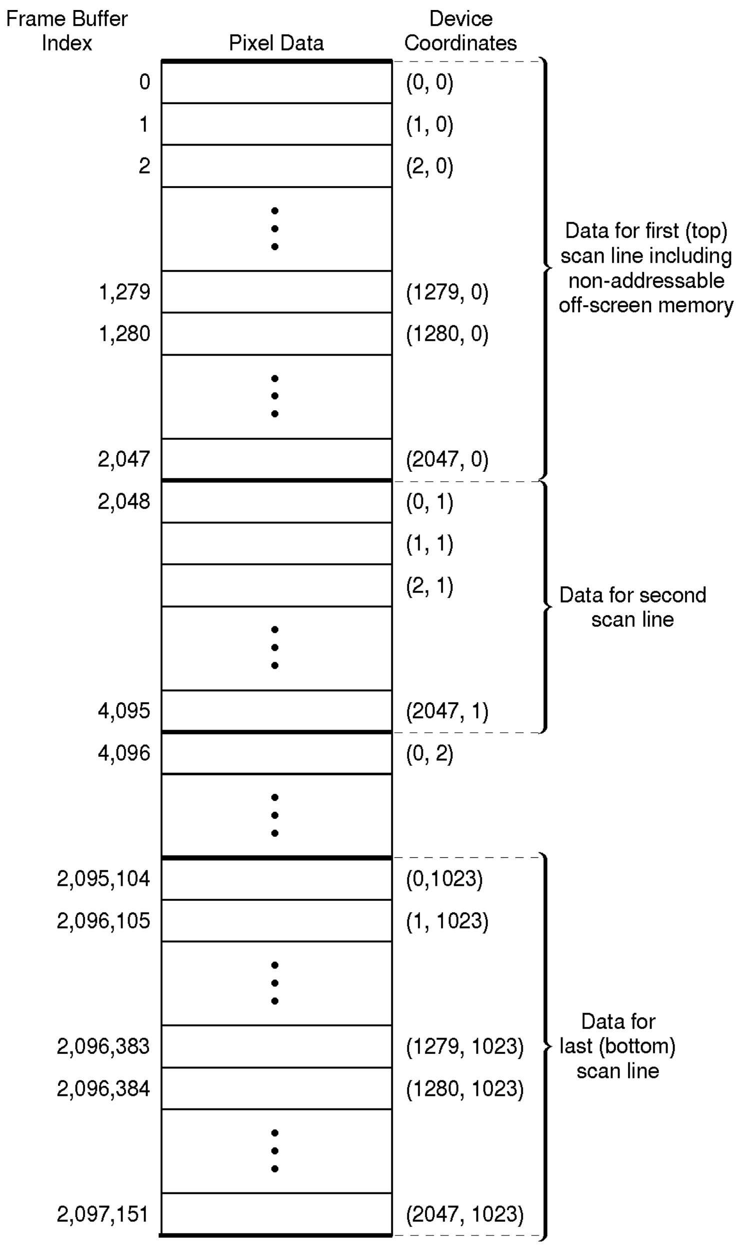

Figure 4-4. Frame Buffer Mapping in Memory

The frame buffer organization is essentially the same for all the devices except for the number of banks. The integrated graphics systems (HP 705/710/715/725) have one bank of 8 planes. The GRX, CRX and Dual CRX have two banks of 8 planes (one for each buffer), and the CRX-24 and CRX-24Z have three banks of 8 planes (one 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.

For normal (non-raw) `block_read` and `block_write` operations to the image planes, the data is in all eight bits of each byte. The Z-buffer for the CRX-24Z is 24-bits deep. For the integrated graphics systems color, CRX, Dual CRX or CRX-24 with PowerShade, the Z-buffer is 16-bits deep. The default for reading the Z-buffer is always 32-bits per pixel. The Z-buffer data is shifted to be either leftmost 24 or 16 bits within the 32-bit word. The `raw` parameter to `block_read` and `block_write` must be set to true in order to read or write to the Z-buffer. Using bank 3 in the `bank_switch` command on the CRX-24 and CRX-24Z, or bank 2 on the other displays, selects the Z-buffer for reads or writes.

Unlike the frame buffer, the Z-buffer data is contiguous. The CRX-24Z Z-buffer is always 1280x1024 and word 1280 is the leftmost word of the second scanline. For either the integrated graphics systems color, CRX, Dual CRX or CRX-24, the Z-buffer is 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.

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 4-4. Supported Frame Buffer Configurations

Number of Planes	HP705/705 710/725	GRX,CRX, Dual CRX	CRX-24, CRX-24Z
8	CMAP_NORMAL CMAP_FULL CMAP_MONOTONIC	CMAP_NORMAL CMAP_FULL CMAP_MONOTONIC	CMAP_NORMAL CMAP_FULL CMAP_MONOTONIC
8/8		CMAP_NORMAL CMAP_FULL CMAP_MONOTONIC	CMAP_NORMAL CMAP_FULL CMAP_MONOTONIC
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 the GRX, CRX, Dual CRX, and CRX-24, or a depth of 12 when in `CMAP_FULL` mode on CRX-24. 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.

X Windows

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

Per-Window Double-Buffering

The GRX, CRX, Dual CRX, CRX-24 and CRX-24Z all support double buffering on a per-window basis. The GRX, CRX and Dual CRX support up to 8/8 planes double-buffered for each of the Starbase color map modes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`). In the image planes, the CRX-24 and the CRX-24Z support the above mentioned modes and also 12/12 planes double-buffered in direct color mode (`CMAP_FULL`). Overlay plane windows do not support any double buffering. Any X11 library drawing routines will render to the currently visible buffer of a window that has double-buffering enabled.

Deeper Overlay Planes

To remove the need for running the X server in the image planes, the number of planes in the overlays has been increased to 8 on CRX-24 and CRX-24Z. (Previous devices had only 4 planes.)

Color Maps

`SB_X_SHARED_CMAP` (environment variable) allows for better sharing of the color map between X11 and Starbase graphics for the low-end (HP 705/710/715/725) and mid-range (GRX, CRX and Dual CRX) devices. Enabling the environment variable greatly reduces the technicolor effect that occurs between simultaneously active applications by providing a default color map that is usable by both X and Starbase. Some cells of the `SB_X_SHARED_CMAP` can be modified if necessary; however, this may cause the technicolor effect when contention with another application happens.

On the high-end devices (CRX-24 and CRX-24Z), the technicolor effect is less likely because there are four hardware color maps available (a single 24-plane direct color color map and three other 8-plane `PseudoColor` color maps). The environment variable (`SB_X_SHARED_CMAP`) is ignored on these devices. Overlay planes use yet another separate color map.

See the section *If You See the Technicolor Effect* for more details on the environment variable.

Backing Store

The devices all support backing-store (also known as retained raster). The backing-store feature allows a window being rendered to by a 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 menu is popped on top of a window containing a complete image,the window system will save the contents of the window before displaying the menu. Then, when the menu 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. There are, however, some exceptions to this. Backing store with 24-plane visuals is not supported. Backing store cannot be enabled for the CRX-24Z accelerator. Certain gescape operations accessing device-dependent features don't work with backing store. See the *Exceptions to Gescapes Support* section in this chapter for more information. Also, mixing Xlib rendering with Starbase rendering to an 8/8 window will cause backing store contents to be lost.

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

Overlay Planes/Image Planes

For the high-end devices (CRX-24 and CRX-24Z), both 8 overlay planes and 24-image planes are provided. Generally speaking, the overlay planes can be thought of as primarily used for the user-interface and the image planes for graphics (either generated by Starbase or Xlib). The X server uses the overlay planes. Graphics applications should request image plane visuals. The *xwcreate* call creates a depth 8 image plane visual by default. HP VMX is used as the exclusive Starbase driver for the overlay planes on the CRX-family of devices (see the section "HP VMX: The CRX-family Overlay Plane Driver" in the chapter "HP VMX Device Driver").

These two sets of planes exist with little interaction. For example, when a menu in the overlay planes occludes a window in the image planes, no "expose event" occurs because the image frame buffer contents are not affected. Only other windows in the image planes can generate "expose events". This happens less frequently. Also, when a window in the overlay planes occludes another window in the image planes, there is no performance degradation caused by the occlusion.

Color map index 255 for the overlay planes is the transparent color on CRX-24. Any overlay pixels written with this color will not display so you can see through to the image plane. Attempts to modify the color map for this index will have no effect.

Expect the following side effect when rendering to the overlay planes: Since Starbase echos for image plane windows reside in the overlay planes, another overlay window will obscure these, even if that window is transparent. If this is not desirable, you can place the echo in an overlay plane child window instead, accessing it as a separately gopened device. Stacking will work correctly with respect to other overlay windows. The X window stacking semantics will give you the desired effect.

Note See the *Starbase Graphics Techniques* manual section on the `SERVER_OVERLAY_VISUALS` property for information on how to determine the visual(s) (if any) that contain the transparent color on a device.

Supported X Windows Visuals

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*.

X11 Cursor

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

How to Read the Supported Visuals Table

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

Table 4-5. Supported Visuals

Device	Depth	Visual Class	Backing Store		Starbase Doublebuffer
			Xlib	Starbase	
705/710/ 715 Gray	8	GrayScale	●	●	[1]
705/710/ 715/725 Color	8	PseudoColor	●	●	[1]
GRX	8	GrayScale	[2]	●	8/8[1]
CRX, Dual CRX	8	PseudoColor	[2]	●	8/8[1]
CRX-24, CRX-24Z Image Planes	8 12 24	PseudoColor DirectColor DirectColor	[2] [3] [3]	● No No	8/8[1] 12/12 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.

2 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. After such a window is made visible and then obscured again, the X11 server will start to maintain backing store again.

3 Xlib backing store supported except for Direct Hardware Access (DHA) windows.

Starbase Echos

Implementation

The hpgcrx device driver implements the X cursor in hardware and all Starbase echo types in software, but utilizes the hardware as best as possible.

The GRX, CRX and Dual CRX devices store Starbase echos in the opposite buffer from the one being rendered to. Echo shares the image plane using an XOR non-destructive replace.

The CRX-24 and CRX-24Z device drivers always XOR the echo into the overlay planes when rendering to either the image or overlay planes.

The HP 705/710/715/725 displays also XOR the echo into the same planes that are being rendered to.

Exceptions

CRX, Dual CRX, GRX

Starbase echos have the benefit of overlay plane functionality even though there are no overlay planes. The echo automatically resides in the bank where normal Starbase rendering is not in operation, that is, the bank that is not enabled for writing. Even when the displayed bank is the same as the write-enabled bank, this function can be used.

On the CRX, Dual CRX and GRX, there are two situations where this can cause problems:

1. If there are 2 gopens to the same window, and each is writing to a different bank.

   There is no place to put the echos. One of the gopens will run slower than normal because of the overhead of removing and replacing the echo around each rendering operation.

2. XORing the echo into the non-displayed bank.

   When the bank being displayed is the same as the bank being written to, the cursors are put in the non-displayed bank. This is okay since the bank being displayed can be selected on a per pixel basis. The side effect is that the cursor will be `XOR`ed with the image in the non-displayed bank, which could cause some unexpected color combinations.

CRX-24, CRX-24Z

The CRX-24 device driver, when rendering to the image planes, stores Starbase echos in the overlay planes for the CRX-24 so they are never in the way when rendering to the image planes. Visually, they are combined with the image by obscuring it, rather than by an `XOR` operation.

On the CRX-24 and CRX-24Z, there are some side effects that can cause problems:

- Windows in the overlay planes can cause the echo to be obscured, even if the window contains transparent color.
- A common operation may be to have a window in the image planes with a transparent child window in the overlay planes. In this case, put the echo in the overlay-plane child window rather than the image-plane parent window. For device-independent behavior in this situation, use the `inquire_capabilities()` Starbase entry point. If the `IC_TRANS_WIN_IMAGE_CURSOR` bit of the `CONTROL_CAPABILITIES` byte is set, put the echo in the image-plane window. If that bit is *not* set (on the CRX-24 and CRX-24Z the bit will not be set), put the echo in the overlay-plane child window.

To Set Up the Device

The `mknod` command (see `mknod(8)` in the *HP-UX Reference* manual), creates a device special file that is used to communicate between the computer and the display device. The name of this device special file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create device special files, they are normally created by the system administrator.

Although device special files may be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the device special file, however the name that is suggested for the default device is `/dev/crt` for the image planes and `/dev/ocrt` for the overlay planes.

To Create Special Device Files (mknod)

For Series 400

For an SPU with one SGC interface slot, a sample `mknod` entry would be:

```
/etc/mknod /dev/crt c 12 0x010300
```

For the Series 700

For an SPU with only one SGC interface slot, a sample `mknod` entry for the image planes would be:

```
/etc/mknod /dev/crt c 12 0x100000
```

For an SPU with two SGC interface slots, a sample `mknod` entry for the image planes on the device in the second slot would be:

```
/etc/mknod /dev/crt1 c 12 0x000000
```

To access the overlay planes on CRX-24, the least significant bit of the minor number should be 1. A sample `mknod` entry would be:

```
/etc/mknod /dev/ocrt c 12 0x100001
```

Note No separate device special file is necessary for CRX-24Z since the accelerator shares the SGC slot with the CRX-24.

For Dual CRX

When using the Dual CRX, examples of `mknod` entries for the single slot would be:

```
/etc/mknod /dev/crt0 c 12 0x100000
/etc/mknod /dev/crt1 c 12 0x100004
```

When using a second Dual CRX, you can use the following for the second slot:

```
/etc/mknod /dev/crt2 c 12 0x000000
/etc/mknod /dev/crt3 c 12 0x000004
```

To configure the devices in the `/dev` directory, refer to the following sample entries:

Sample for ONE Dual CRX card in an HP 720 or 730:

```
/etc/mknod /dev/crt0 c 12 0x100000
/etc/mknod /dev/crt1 c 12 0x100004
```

Sample for TWO Dual CRX cards in an HP 750:

```
/etc/mknod /dev/crt0 c 12 0x100000     Slot 0
/etc/mknod /dev/crt1 c 12 0x100004

/etc/mknod /dev/crt2 c 12 0x000000     Slot 1
/ect/mknod /dev/crt3 c 12 0x000004
```

Note For physical locations of the above slots, refer to the Dual CRX Installation Notes or the Series 700 CE Hardware Manual.

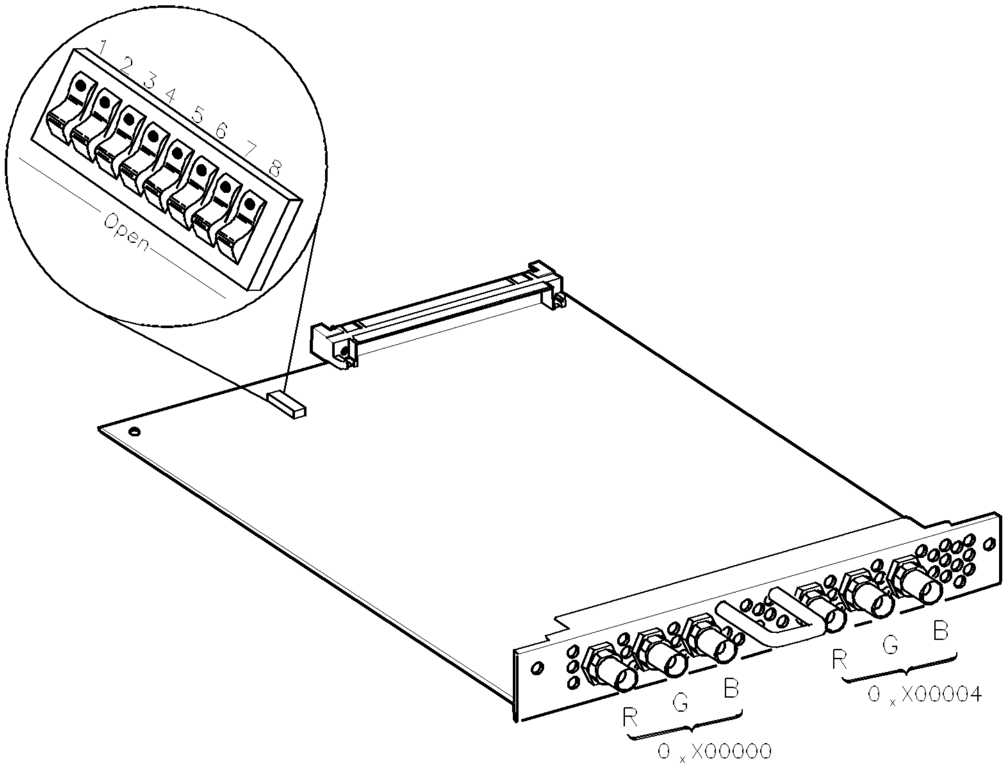

Figure 4-5. Dual CRX - Dual Monitor Graphics Card

To Compile and Link with the Device Driver

For Shared Libraries

The compiler driver programs (`cc, fc, pc`) link with shared libraries by default. The shared device driver is the file named `libddgcrx.sl` in the `/usr/lib` directory. The same device driver is used for all of these devices. Starbase will explicitly load the device driver at run time when you compile and link with the Starbase shared library `/usr/lib/libsb.sl`, or use the `-lsb` option. This loading occurs at `gopen`(3G) time.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter for details.

For Archive Libraries

The archive device driver is located in the `/usr/lib` directory with the file name `libddgcrx.a`. The same device driver is used for all these devices.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libddgcrx.a`
2. an appropriate relative path name
3. the `-lddgcrx` option with the `LDOPTS` environmental variable exported and set to `-a archive`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -lddgcrx -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -lddgcrx -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

or for Pascal, use:

```
pc example.p -lddgcrx -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

To Open and Initialize the Device for Output

Syntax Examples

C programs:

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

FORTRAN77 programs:

```
fildes = gopen('/dev/screen/window'//char(0), OUTDEV,\
                char(0), INIT)
```

Pascal programs:

```
fildes := gopen('/dev/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 special device file created by the `mknod` command as specified in the last section. For example, `/dev/screen/window`.
- *Kind* - This parameter must be `OUTDEV`, unless used for a graphics window, in which case `OUTINDEV` may be used.
- *Driver* - The character representation of the driver type. This parameter may be set to `NULL` for linking shared or archive libraries; `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:

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

Note The name used does not affect the operation of the driver. Other accepted names are: `CRX`, `crx`, `hpA1659A`, `hpa1659a`, `GRX`, `hpA1924A`, `hpa1924a`, `crx24`, `hpA1439A`, `hpa1439a`, `crx24z`, `hpA1454A`, `hpa1454a`, `hp710`.

- *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 X 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. Frame buffer 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 (for CRX-24 with PowerShade and CRX-24Z)
  - `RESET_DEVICE` - Same as INIT. In addition, hardware is reset.
  - `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. Shading is supported only through PowerShade for the `hpgcrx` devices.
  - `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.

This mode is provided for compatibility of integer precision with previous devices. `INT_XFORM` will 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`.

- □ `UNACCELERATED` - Force the `hpgcrx` driver to disable the CRX-24Z accelerator.
- □ `ACCELERATED` - Force the driver to enable the CRX-24Z accelerator if present (default).

Note `SPOOLED` and `MODEL_XFORM` flag bits have no device-dependent effects.

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location $(0,0)$ is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is $(1279,1023)$.

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

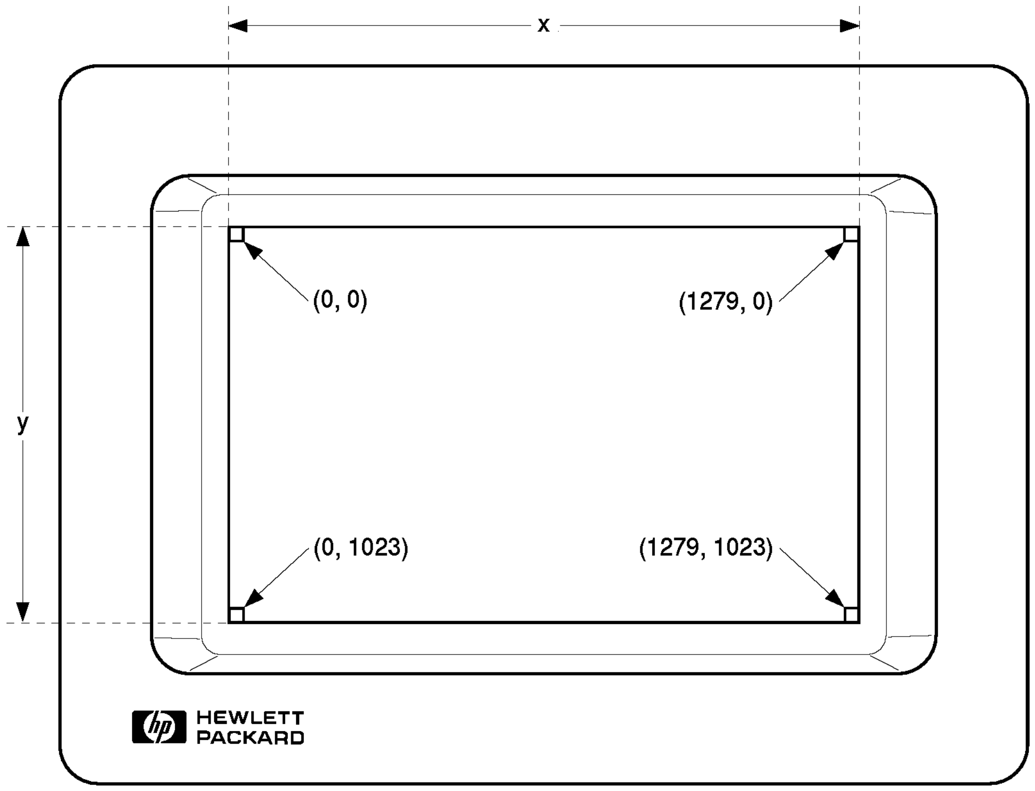

Figure 4-6. Device Coordinates

The HP 705, HP 710, 715 and HP 725 layouts are similar to the above except that the device coordinates for the corners are different because of the difference between the 1280x1024 and 1024x768 resolutions.

4

If You See the Technicolor Effect

You can reduce the visual effects of color map thrashing, also called the *technicolor* or *kaleidescope effect*, and thus reduce distractions due to color map changes. Set the environment variable `SB_X_SHARED_CMAP` to true to use a shared color map between Starbase and X.

The Starbase device driver and the X server for the GRX, CRX, dual CRX and HP 705/710/715/725 devices support the environment variable which indicates the use of a shared or cooperative color map definition. The environment variable allows sharing of the color map resource by Starbase applications with other X clients. It eliminates the creation and installation of different color maps for each different Starbase application being run.

For generic X Windows

If you are using `x11start`, make sure you have the environment variable `SB_X_SHARED_CMAP` set before you execute x11start:

```
export SB_X_SHARED_CMAP=true
```

The best way to do this is to include it in your `$HOME/.profile`.

For HP VUE

If you are using HP VUE (Visual User Environment) add the following line to the end of your `/usr/vue/config/Xconfig` file:

```
Vuelogin*environment:SB_X_SHARED_CMAP=true
```

The `Xconfig` file may contain commented out entries for some of the more popular resources, including "environment". Simply find the line containing "environment", add the appropriate value and uncomment the line.

When using HP VUE with `SB_X_SHARED_CMAP` set, there is one other resource of interest. The HP VUE *colorUse resource in

```
/usr/vue/config/sys.resources
```

can be set to use less colors if it is desired that HP VUE not use up all the unallocated color map cells. The default value on GRX and CRX is `HIGH_COLOR`.

Other values are `MEDIUM_COLOR` and `LOW_COLOR`. The full range of accepted values for this resource can be found in HP VUE documentation.

Please note that changing the `/usr/vue/config/sys.resources` file will only affect users who are created after the change.

To change this setting for an existing user, you can simple change the `*colorUse` setting via a dialog in the HP VUE Style Manager:

- log in to VUE
- select the Style Manager icon from the front panel
- select Color
- select "HP VUE Color Use"
- select the desired color use (High, Medium, Low, B/W, Default)
- select OK

Color Map Sharing Starbase and X Windows

On HP 705/710/715/725, CRX, and Dual CRX applications, you need to use either CMAP_NORMAL without modifying the color map (e.g. define_color_table) or CMAP_FULL to avoid the technicolor effect. On the GRX, applications can note that the device is grayscale (use inquire_capabilities) and make use of only CMAP_MONOTONIC.

For Starbase applications, the SB_X_SHARED_CMAP environment variable enables use of the initial eight entries of the Starbase default CMAP_NORMAL color map. It also provides for CMAP_FULL on the CRX or CMAP_MONOTONIC on the GRX, as well as reserving some color map entries for "X" clients to use. If an X client inquires the default color map for the root window, this shared color map is returned.

When the SB_X_SHARED_CMAP environment variable is set on a grayscale device, the default color map mode is CMAP_MONOTONIC (when gopen mode is INIT) since this is the only mode that can be shared.

When SB_X_SHARED_CMAP is set, double_buffer will return eight planes regardless of the number requested since only 8/8 double-buffering is allowed when SB_X_SHARED_CMAP is set.

If the the behavior with SB_X_SHARED_CMAP is unacceptable for a specific application, unset the environment variable prior to invoking the application or calling gopen. See putenv(3C). This will allow the normal color map functionality, but the technicolor effect will be visible whenever windows used by this application have the color map focus.

The following tables describe the default color map for color devices when SB_X_SHARED_CMAP is set and color map mode is CMAP_NORMAL or CMAP_FULL.

Table 4-6. HP 705/710/715/725 Default Shared Color Map

Index	Type of Cell	Comments
0 . . . 7	Read-Only	same as default Starbase color map
8 . . . 39	Read/Write	allocated dynamically by X server
40 . . . 255	Read-Only[1]	standard 6|6|6 (ramps of six levels each of red, green, blue) XA_RGB_DEFAULT_MAP property defined.

1 Can be made writable by setting SB_X_SHARED_CMAP=READ_ONLY_PRIMARIES.

The following is the GRX default color map when SB_X_SHARED_CMAP is set and the grayscale color map mode is set to CMAP_MONOTONIC.

Table 4-7. GRX Default Color Map

Index	Type of Cell	Comments
0 . . . 255	Read-Only	grayscale ramp Starbase CMAP_MONOTONIC StaticGray visual class in X XA_RGB_DEFAULT_MAP property defined

Table 4-8. CRX Default Shared Color Map

Index	Type of Cell	Comments
0 ... 7	Read-Only	same as default Starbase color map
8 ... 39	Read/Write	allocated dynamically by X server
40 ... 255	Read-Only[1]	modified 6|6|6[2] (ramps of 6 levels each of red, green, blue arranged to support special shading hardware)

1 Can be made writable by setting `SB_X_SHARED_CMAP=READ_ONLY_PRIMARIES`.

2 Can be set to standard 6|6|6 color ramp by setting `SB_X_SHARED_CMAP = XA_RGB_DEFAULT` (read-only).

Two notes on the X server's use of the `SB_X_SHARED_CMAP` environment variable:

- These tables apply to the default color map of the root visual. Other color maps in the root visual will be entirely read/write, initialized with a copy of the default color map's current values.
- The X server sets the `XA_RGB_DEFAULT_MAP` only if a standard read-only ramp is defined in entries 40-255. A Xlib program can read a description of the ramp and the ID of the default color map implementing the same by calling `XGetStandardColormap`.

The `SB_X_SHARED_CMAP` environment variable is not supported on the CRX-24 or CRX-24Z. It is not needed because there are three PseudoColor color maps in the image planes. Use of these three color maps on CRX-24 is encouraged in two ways:

1. *xwcreate* - depth 8 (or no depth specification) by default creates a window in the image planes. (To force a window into the overly, use `xwcreate_overlay`.)
2. With Xlib programmatic access, the image plane depth 8 visual will be the first depth 8 visual returned by `XGetVisualInfo()`.

TRANSPARENT_IN_OVERLAY_VISUAL (CRX-24, CRX-24Z)

The default number of entries in the visual for the CRX-24's overlay planes is 255. Entry 255 is excluded because its value is hard-coded to transparent (that is, show the image planes).

While this is the most correct choice, it does have two consequences for clients running the the CRX-24's overlay planes (the default visual):

- Clients attempting to allocate 256 entires will not have their request granted.
- Clients requesting (via `XAllocNamedColor`) the `rgb.txt` value of "Transparent" will not be returned entry 255.

If you are using `x11start`, you can change this behavior by setting the `CRX24_COUNT_TRANSPARENT_IN_OVERLAY_VISUAL` environment variable to `TRUE`.

The example below works if you are running `ksh`:

```
export CRX24_COUNT_TRANSPARENT_IN_OVERLAY_VISUAL=TRUE
```

If you are using HP VUE (Visual User Environment), add the following (as one line) to the end of your `usr/vue/config/Xconfig`:

```
Vuelogin*environment:\
CRX24_COUNT_TRANSPARENT_IN_OVERLAY_VISUAL=TRUE
```

With this variable set, the X server will:

- Specify that the overlay visual has 256 entries. (On a call to `XGetVisualInfo()`, this overlay visual is listed after the 8-plane PseudoColor visual in the image planes.)
- Create the default color map with entry 255 pre-allocated to "Transparent". A client calling `XAllocNamedColor` for entry "Transparent" in the default color map will be returned entry 255.
- For all other color maps, return all 256 entries as allocable, but issue a warning message:"Warning: `XCreateColormap` is creating 256 entry cmaps in overlay visual. Though allocable, entry 255 is hard-coded to transparency." This warning is issued once per server execution.

CMAP_FULL Color Map

When the entire color map is available, Starbase uses standard 3:3:2 direct color (CMAP_FULL mode) in 8-plane visuals. With the SB_X_SHARED_CMAP set, only 216 entries of the color map are available for direct color support.

Starbase defines these entries as follows:

The standard 3:3:2 indicates that three bits are set aside for red, three for green and two for blue. This is the same color map that is used on other 8-plane devices.

The standard 6|6|6 means that the color map contains all combinations of six *levels* of red, green and blue. This yields a total of 6x6x6 = 216 colors. Color *levels* are evenly spaced from *black* (i.e. zero intensity) to *white* (i.e. full intensity) using the following values: { 0.0, 0.2, 0.4, 0.6, 0.8, 1.0 }.

The entries between 40 and 255, inclusive, are used for the direct color map with SB_X_SHARED_CMAP enabled. They are defined according to the following algorithm:

```
int index;
CMAP shared_color_map[256];
float red_level,
      green_level,
      blue_level;
. . .

index = 40;
 for (red_level = 0.0; red_level <= 1.0; red_level += 0.2)
   for (green_level = 0.0; green_level <= 1.0; green_level += 0.2)
     for (blue_level = 0.0; blue_level <= 1.0; blue_level += 0.2)
       {
        shared_color_map [index].red_component = red_level;
        shared_color_map [index].green_component = green_level;
        shared_color_map [index].blue_component = blue_level;
        index++;
       }
```

This is known as the standard 6|6|6 organization for the color map.

The CRX, Dual CRX and integrated graphics systems (HP 705/710/715/725) color devices provide support for both the 3:3:2 and 6|6|6 CMAP_FULL color maps. The standard 3:3:2 color map is supported in raw mode and in X11 windows when the SB_X_SHARED_CMAP mode is off. The 6|6|6 color map mode is available in X11 Windows when the SB_X_SHARED_CMAP mode is on.

Note The hpgcrx driver supports CMAP_FULL mode in depth 24 windows on the CRX-24. It also supports CMAP_FULL mode in depth 8 windows on the same device for compatibility with other devices. Performance is better with depth 24 windows. We strongly discourage use of CMAP_FULL in depth 8 windows on the CRX-24.

CMAP_FULL Translations

Some devices supported by the hpgcrx driver use a special color map organization internally to support hardware dithering. The modified color map is used on the GRX, CRX and Dual CRX. This hardware color map organization is known as modified 3:3:2 or modified 6|6|6 corresponding to the standard organizations described above. Under normal circumstances, this internal modified color map is transparent to the user.

The CRX-24, the CRX-24Z and the integrated graphics systems color configurations use the standard color maps.

On those devices that use the modified color maps in hardware, there are compatibility issues for some applications. These potential issues effect Starbase only in CMAP_FULL mode.

In most cases, Starbase hides the modified organization of the color map from the applications. However, there are some exceptions to this.

Starbase hides the color map organization by implementing a translation layer (SW_CMAP_FULL translation), where necessary, to convert from the standard 3:3:2 to modified 3:3:2, or from a standard 6|6|6 to the modified 6|6|6 on GRX/CRX. This translation layer is enabled or disabled with the GCRX_SW_CMAP_FULL gescape or when the *raw* parameter passed into block_write(3G) and block_read is set to *TRUE*. The default at the time of gopen is to enable translation.

The following sections describe the effects of this translation layer on the Starbase semantics.

Block Read and Block Write

The `block_read` and `block_write` calls hide the color map organization when enabling `GCRX_SW_CMAP_FULL` translation. The performance implications of `GCRX_SW_CMAP_FULL` translation mainly apply to these two routines. If performance is critical in this area, disable the translation with the `GCRX_SW_CMAP_FULL` gescape or the *raw* parameter, and use the modified color map organization.

Index Color Attributes

Entry points which take a color attribute as a color map index value hide the color map organization if `GCRX_SW_CMAP_FULL` translation is enabled. This includes the `define_color_table`(3G) call as well as the following:

```
background_color_index
bf_fill_color_index
bf_perimeter_color_index
depth_cue_color_index
fill_color_index
highlight_color_index
inquire_color_table
line_color_index
marker_color_index
perimeter_color_index
text_color_index
```

Plane-oriented Operations

The standard 3:3:2 organization is plane-oriented, and as such, Starbase functions that operate on frame buffer planes are more predictable with the standard 3:3:2 organization than either the CRX modified 3:3:2 or any 6|6|6 color map organization. Examples of such Starbase functions are `drawing_mode`, `write_enable`, and `display_enable` Starbase echos which use *XOR*.

These display control functions (via `display_enable`, `write_enable` and `drawing_mode`) do not hide the reorganized CRX color map organization.

If you need display control functionality, turn off the translation with the `GCRX_SW_CMAP_FULL` gescape and work with the modified color map organization.

Cursors

The `define_raster_echo`(3G) call hides the modified CRX color map organization when `SW_CMAP_FULL` translation is enabled. Vector cursors do not hide the color map organization. (There is minimal impact to the resulting cursor display.)

Pattern Fills

The `pattern_define` call and the pattern-defining gescapes (`GR2D_FILL_PATTERN` and `R_DEF_FILL_PAT`) hide the modified color map organization when the `GCRX_SW_CMAP_FULL` translation is in effect.

Background and Clear Control

When switching to or from `CMAP_FULL` using `shade_mode`(3G) with `INIT`, the CRX/GRX devices clear the entire display. The frame buffer is cleared to all zeros at gopen time when `INIT` is specified. After gopen, if the application changes the clear control to something less than the full display, and switches into `CMAP_FULL` with the modified 3:3:2 definition, the areas outside of the clear control will turn white since index 0 of the modified 3:3:2 color map is defined as red=green=blue=255.

When `gopen` is called with `INIT`, the default is to clear the display if `shade_mode` is called with `INIT`. When `gopen` is called without `INIT`, the default is to leave the frame buffer as is. Clearing the display when switching to and from `CMAP_FULL` with `INIT` is controlled with the `GCRX_SW_CMAP_FULL` gescape.

Translation from Standard to Modified Color Map Indices

For applications that need to bypass the Starbase color map translation layer, the translation required is a simple table look-up of the standard 3:3:2 value to get the modified 3:3:2 value. The same is true for translation from standard 6|6|6 to modified 6|6|6.

The following tables show this mapping of color map indices.

CMAP_FULL Color Map Index Translation

Table 4-9. Translation Table for standard 6|6|6 values to modified 6|6|6 values:

Index	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
10	10	11	12	13	14	15	16	17	18	19
20	20	21	22	23	24	25	26	27	28	29
30	30	31	32	33	34	35	36	37	38	39
40	255	251	191	187	127	123	253	249	189	185
50	125	121	239	235	175	171	111	107	237	233
60	173	169	109	105	223	219	159	155	95	91
70	221	217	157	153	93	89	254	250	190	186
80	126	122	252	248	188	184	124	120	238	234
90	174	170	110	106	236	232	172	168	108	104
100	222	218	158	154	94	90	220	216	156	152
110	92	88	247	243	183	179	119	115	245	241
120	181	177	117	113	231	227	167	163	103	99
130	229	225	165	161	101	97	215	211	151	147

Table 4-9. Translation Table for standard 6|6|6 values to modified 6|6|6 values: (continued)

Index	0	1	2	3	4	5	6	7	8	9
140	87	83	213	209	149	145	85	81	246	242
150	182	178	118	114	244	240	180	176	116	112
160	230	226	166	162	102	98	228	224	164	160
170	100	96	214	210	150	146	86	82	212	208
180	148	144	84	80	207	203	143	139	79	75
190	205	201	141	137	77	73	199	195	135	131
200	71	67	197	193	133	129	69	65	63	59
210	55	51	47	43	61	57	53	49	45	41
220	206	202	142	138	78	74	204	200	140	136
230	76	72	198	194	134	130	70	66	196	192
240	132	128	68	64	62	58	54	50	46	42
250	60	56	52	48	44	40				

Code example:

To create an array initialized to the above values:

```
unsigned char mod_666_table[256]  = /*values from above table*/;
```

Access the table as follows:

```
std_666_index = floor(float_red*5)*36
                + floor(float_green*5)*6
                + floor(float_blue*5)
                + 40;

mod_666_index = mod_666_table[std_666_index];
```

Table 4-10.
Translation Table for standard 3:3:2 values to modified 3:3:2 values

Index	0	1	2	3	4	5	6	7	8	9
0	255	251	223	219	253	249	221	217	239	235
10	207	203	237	233	205	201	127	123	95	91
20	125	121	93	89	111	107	79	75	109	105
30	77	73	254	250	222	218	252	248	220	216
40	238	234	206	202	236	232	204	200	126	122
50	94	90	124	120	92	88	110	106	78	74
60	108	104	76	72	247	243	215	211	245	241
70	213	209	231	227	199	195	229	225	197	193
80	119	115	87	83	117	113	85	81	103	99
90	71	67	101	97	69	65	246	242	214	210
100	244	240	212	208	230	226	198	194	228	224
110	196	192	118	114	86	82	116	112	84	80
120	102	98	70	66	100	96	68	64	191	187
130	159	155	189	185	157	153	175	171	143	139
140	173	169	141	137	63	59	31	27	61	57
150	29	25	47	43	15	11	45	41	13	9
160	190	186	158	154	188	184	156	152	174	170
170	142	138	172	168	140	136	62	58	30	26
180	60	56	28	24	46	42	14	10	44	40

Table 4-10. Translation Table for standard 3:3:2 values to modified 3:3:2 values (continued)

Index	0	1	2	3	4	5	6	7	8	9
190	12	8	183	179	151	147	181	177	149	145
200	167	163	135	131	165	161	133	129	55	51
210	23	19	53	49	21	17	39	35	7	3
220	37	33	5	1	182	178	150	146	180	176
230	148	144	166	162	134	130	164	160	132	128
240	54	50	22	18	52	48	20	16	38	34
250	6	2	36	32	4	0				

Code example:

To create an array initialized to the above values:

```
unsigned char mod_332_table[256]  = /*values from above table*/;
```

Access the table as follows:

```
std_332 index = (float_red*7)<<5
                + (float_green*7)<<2
                + (float_blue*3);

mod_332_index = mod_332_table[std_332_index];
```

Starbase Functionality

Calls not Supported

The hpgcrx 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 (i.e. they are no-ops).

```
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
```

PowerShade or CRX-24Z on CRX-24

The following calls are not supported when using the CRX-24Z or PowerShade 3D Surfaces Software with CRX-24:

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

In addition, `line-filter` and `perimeter_filter` are *not* supported on the CRX-24.

Using PowerShade on CRX, Dual CRX or High Resolution Integrated Graphics Color

The following calls are not supported when using PowerShade on the CRX, Dual CRX or High Resolution Integrated Graphics (HP 705/710/715/725) Color:

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

Note PowerShade is not supported on the Series 400 or on the grayscale configurations.

Conditional Support of Starbase Calls

The following calls are supported with the listed exceptions:

`alpha_transparency`	Alpha transparency is supported only on the CRX-24Z. Alpha only applies to filled areas such as polygons, quadrilateral meshes, triangular strips, and spline surfaces. Alpha per vertex is not supported. 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 CRX-24Z does not support alpha transparency with attenuation. (See the manpage on alpha_transparency for parameters in the *Starbase Reference Manual*).
`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 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 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.

`line_filter`
`perimeter_filter`

Antialiasing is supported only on the CRX-24Z. Antialiasing for this device applies only to floating point vectors. Device coordinate primitives do not use antialiasing. The antialiasing features are also limited to the `CMAP_FULL` color map mode in the 12/12 or 24-plane configurations.

The CRX-24Z has two antialiasing 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 three pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according the the formula: *pixel_color*= (*new_pixel* $\times$ α) + (*old_pixel* $\times$ (1 - α));
	`2` Anti-aliasing enabled, all vectors have three pixel wide output. Pixel values are multiplied by the alpha value, but are NOT blended with the background. *pixel_color* = (*new_pixel* $\times$ α);
`pattern_define`	For the CRX-24Z, 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.
`screenpr`	Because of per-window double-buffering and multiple color maps, this utility is only supported for printing one window at a time. Attempts to print pixels outside this one window may result in a wrong color or pixels from the wrong buffer being printed.
`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`	The ⟨*use*⟩ parameter must be zero. Any extra coordinate info will be ignored. If using PowerShade software, `vertex_format` is fully functional.

`with_data`	`partial_polygon_with_data3d`
	`polygon_with_data3d`
	`polyhedron_with_data`
	`polyline_with_data3d`
	`polymarker_with_data3d`
	`quadrilateral_mesh_with_data`
	`triangle_strip_with_data`
	Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

`write_enable`	Due to hardware limitations on the CRX-24, certain `write_enable` masks are not supported. Only those masks whose bits 0-3 are the same as bits 4-7 are supported.

Supported Gescapes

The hpgcrx driver supports the following gescape operations. Refer to Appendix A of this manual for details on gescapes.

- BLOCK_WRITE_SKIPCOUNT—Specify byte skip count during block write.
- GCRX_PIXEL_REPLICATE—Pan and zoom a raster image.
- GCRX_SW_CMAP_FULL—Control CMAP_FULL color map translation. (0-OFF-ON 3-ON and CLEAR DISPLAY)
- IGNORE_RELEASE—Trigger only when button pressed.
- R_BIT_MASK—Bit mask.
- R_BIT_MODE—Bit mode.
- R_DEF_ECHO_TRANS—Define transparency mask for raster echo.
- R_ECHO_MASK—Turns on echo transparency mask.
- 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.
- READ_COLOR_MAP—Read Color Map.
- SWITCH_SEMAPHORE—Semaphore Control.
- TRIGGER_ON_RELEASE—Trigger only when button is released.

Additional Gescapes for the CRX-24 and CRX-24Z

- 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.
- GAMMA_CORRECTION—Enable/disable gamma correction.
- LINEAR_POLYPOINT—Specify points to be rendered along a line specified in modeling coordinates.
- STEREO—Activate stereo output mode.

Additional Gescapes for the CRX-24Z

- DRAW_POINTS—Select different modes of rounding for rendered points.

Additional Gescapes Supported with PowerShade

- 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).

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 differently on `hpgcrx` than on older devices. It is now implemented in the color map rather than in the frame buffer. For information on the gescape GAMMA_CORRECTION, refer to Appendix A.

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.

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. This gescape is supported for backing store; i.e. the correct data is returned from the retained raster for those parts of the window that are obscured.
R_BIT_MODE	When `block_read` or `block_write` are used with the *raw* parameter set to TRUE, and *raw* mode is enabled, the driver supports bit/pixel frame buffer access to single planes.

5

CRX-48Z Device Driver

High Performance Grayscale and Color Graphics

The `hpcrx48z` driver supports the CRX-48Z graphics accelerator on Series 700 Workstations. Features of the CRX-48Z include:

- 48 image planes
- 24 Z-planes
- 8 overlay planes
- 5 hardware color maps
- Hardware support for generating flat and smooth shaded vectors and polygons used in 3D solids modeling
- Hardware support for rendering anti-aliased vectors
- 72Hz 1280x1024 pixel 19-inch color monitor

The 3D surfaces software, PowerShade (B2156A), is used by the `hpcrx48z` driver to transform and clip graphics primitives. Powershade is also used to convert complex primitives such as spline surfaces and polygons with many vertices into simple primitives which can be rendered by the CRX-48Z hardware. PowerShade is included in all CRX-48Z product bundles.

For More Information

Information in this chapter is device-specific. For more detailed information on the areas listed below, please refer to the noted documents:

- See the *Starbase Graphics Techniques* manual to read about backing store in X Windows.
- Refer to the *Programming on HP-UX* manual to read about linking shared or archive libraries.

Device Description

CRX-48Z (A2091A)

The CRX-48Z device has 48-image planes consisting of two buffers of 24 planes each, and 8-overlay planes. The screen resolution is 1280x1024 pixels. There is no offscreen memory in the frame buffer.

Note The `hpcrx48z` driver can only be used in windows created via the X Window System — the `hpcrx48z` driver does not support *raw* mode access to the device.

The `hpcrx48z` driver supports the following frame buffer configurations:

5

- 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. *Please note that although the CRX-48Z has 48-image planes, it can only display or render to 24-image planes at a time.*

You can select each of the above configurations on a per-window basis. The configuration selected is a function of the depth of the window created and whether the window is in the overlay or image planes.

In the 8-bit indexed configurations, 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 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 planes.

In the 24-bit direct color configuration, each pixel is represented by eight bits per color channel. Double-buffering is achieved by dividing the 48 planes into two buffers of 24 planes. Double buffering of more than 8 planes is promoted to 24/24 double-buffering.

Dithering is used in depth 24 direct color visuals to improve color resolution.

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.

There are five different hardware color maps available on the CRX-48Z, which the X Server manages and allocates on a per-window basis. One of the color maps is shared by all overlay windows, and the remaining four are used by image windows.

Overlay Plane Rendering

Because of hardware restrictions, the `hpcrx48z` device driver cannot render to the overlay planes. If Starbase rendering to the overlay planes is required, the `hpvmx` driver may be used instead. For more information on Starbase rendering to the overlay planes, read the section "HP VMX: The CRX-family Overlay Plane Driver" in the chapter "HP VMX Device Driver."

Although the `hpvmx` driver is a fully functional PowerShade driver, only `CMAP_NORMAL` rendering is supported in the overlay planes of the CRX-48Z device. See below for comments about how the `hpvmx` driver is supported in CRX-48Z image planes. 8/8 double-buffering is also supported in the overlay planes using the `hpvmx` driver.

Note Color map index 255 is the overlay planes is the *transparent* index. Any pixels written with index 255 will display the image plane value at that location.

If an overlay plane window is `gopen`ed with a driver name of `NULL`, the `hpvmx` driver will be used. See Table 3-3, "Driver Selection at gopen" in the chapter "HP VMX Device Driver," for details.

Since the CRX-48Z device has only one color map for the overlay planes, the color map used by Starbase in an overlay window could potentially clash with the window system's color map. To avoid color map contention with the X Window System, overlay plane windows could be `gopen`ed *WITHOUT* doing an `INIT` or `RESET_DEVICE` (see `gopen`(3G) for details). Then, color map entries could be allocated from the X Server like an X client. See the "Developing a Starbase Application" chapter of this manual for an example of how to do this.

In the image planes of the CRX-48Z device, the `hpvmx` driver is a fully functional PowerShade device driver, and supports `CMAP_NORMAL`, `CMAP_MONOTONIC`, and

`CMAP_FULL` rendering, and well as double-buffering. However, the `hpvmx` driver is limited to depth 8 windows (it cannot render to depth 24 windows). The `hpcrx48z` driver does not support backing store. The `hpvmx` driver supports backing store only in the overlay planes.

While the `hpvmx` driver will work in the image planes of the CRX-48Z device, it is not recommended since the `hpcrx48z` device driver has significantly better performance in the image planes.

Optimized 3D Shaded Polygon Performance

The CRX-48Z provides optimized 3D shaded polygon performance. Shaded polygons on the CRX-48Z are highly optimized and very fast in most cases. See the file `/usr/lib/starbase/perf.notes` for on-line PowerShade performance notes on features and performance data. This data is subject to change without notice.

In order to use VMX with PowerShade from a CRX-24Z or CRX-48Z graphics system, you must install the PowerShade software.

The ITE and the CRX-48Z

Since the CRX-48Z is not supported in a non-window environment, ITE keyboard and escape sequences to control the display of ITE text and image graphics are not supported.

Frame Buffer Organization

The image planes on the CRX-48Z consist of two buffers with 24 planes each. Each buffer consists of three banks, one each for red, green, and blue. Each of the 4 image color maps on the CRX-48Z consists of three color tables — one each for red, green, and blue. Regardless of which color map mode is currently being used, the data in each of the banks is used as an index into a color table for that color of bank. For example, the pixel value in the red bank is used as an index into the red color table to determine the intensity of red that should be displayed for that pixel.

As a result of this color map organization, frame buffer configurations that are of depth 8 require that the 8 bits of pixel data be written to all three banks in order for the correct RGB values to be displayed.

A `bank_switch` in depth 8 (`CMAP_NORMAL`) frame buffer configurations selects which of the two 24-plane buffers will be accessed for `block_read` and `block_write`.

Using a bank number of 0 selects the first of the two buffers. This first buffer is the bank currently enabled for writing as selected by the `double_buffer/dbuffer_switch`.Using a bank number of 1 selects the second buffer.

`block_write` duplicates the pixel value to the three banks of the current buffer. `block_read` returns values in the red bank (which should be the same as the values in the green and blue banks).

`bank_switch` in depth 24 (`CMAP_FULL`) frame buffer configurations selects which one of the six 8-plane banks will be accessed for `block_write` and `block_read`.

Please note: In the following table, "current writeable buffer" is defined as the buffer selected by `double_buffer/dbuffer_switch`.

Table 5-1. Bank Selection

wbank	bank selected
0	red bank, current writeable buffer
1	green bank, current writeable buffer
2	blue bank, current writeable buffer
3	red bank, non-writeable buffer
4	green bank, non-writeable buffer
5	blue bank, non-writeable buffer
6	Z-buffer[1]

1 Must use the ZBANK_ACCESS gescape.

The Frame Buffer

Physical Address Space

The physical frame buffer is addressed as 2048x1024 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.

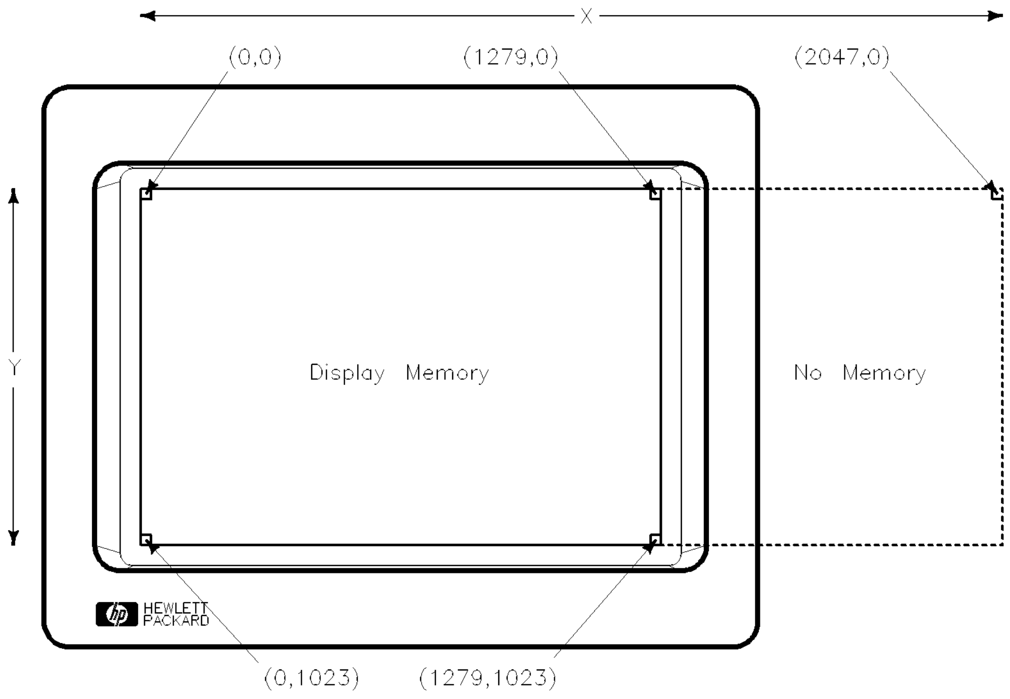

Figure 5-1. Physical Address Space

To Access the Frame Buffer Directly

Programs that use direct frame buffer access need to follow the protocol listed below:

1. Use the `R_LOCK_DEVICE` gescape to gain exclusive access to the device.
2. Flush any buffered primitives by calling `make_picture_current()`.
3. Use the `PLUG_ACCELERATED_PIPELINE` gescape with an argument of 1 to allow access to the frame buffer.
4. Access the frame buffer directly.
5. Use the `PLUG_ACCELERATED_PIPELINE` gescape with an argument of 0 to re-enable Starbase commands.
6. Use the `R_UNLOCK_DEVICE` gescape to allow other processes to now access the device.

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

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 (1023x2048+1279).

Only one bank (8 bits) of the CRX-48Z can be accessed at a time. Use the `bank_switch` call to select a bank to read or write. The bank selected will be the same as the bank selected for `block_read` and `block_write`. If the current rendering mode is 8 planes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, or `CMAP_FULL` 3:3:2), direct frame buffer writes will write the same information to each of the three banks in the appropriate buffer.

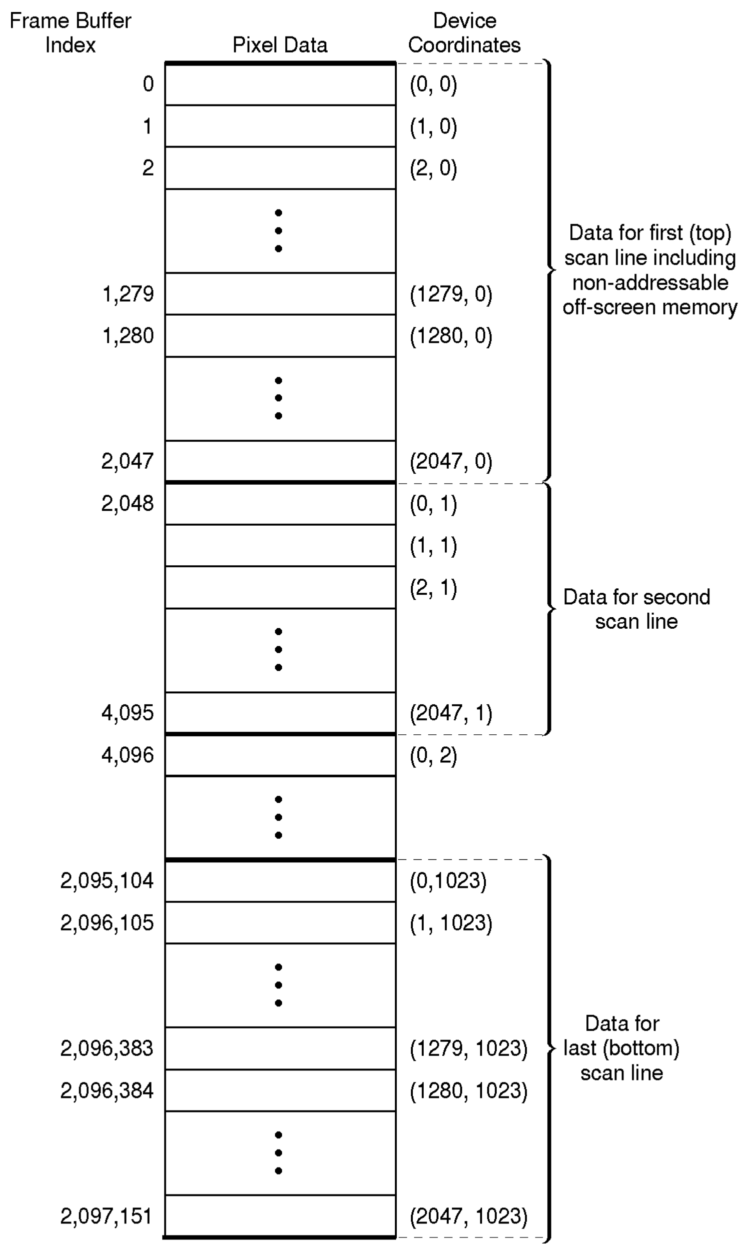

Figure 5-2. Frame Buffer Mapping in Memory

X Windows

The `hpcrx48z` is supported only with X Windows. There is no *raw* device support. The only X server mode supported is combined mode. For more information on X11, please refer to "Using Starbase with the X Window System" in the *Starbase Graphics Techniques* manual.

Overlay Planes/Image Planes

The CRX-48Z has 8 overlay planes and 48 image planes. Generally speaking, the overlay planes can be thought of as primarily used for the user-interface and the image planes for graphics (either generated by Starbase or Xlib). X window managers use the default visual, which is an 8 plane overlay visual. Graphics applications should request image plane visuals. The *xwcreate* command creates a depth 8 image plane visual by default. HP VMX is used as the exclusive Starbase driver for the overlay planes on the CRX-family of devices. Double-buffering, backing store and the `CMAP_NORMAL` color map modes are supported for overlay windows. `CMAP_FULL` and `CMAP_MONOTONIC` color map modes are *not* supported in overlay plane windows. For more information on Starbase overlay planes, read the section "HP VMX: The CRX-family Overlay Plane Driver" in the chapter "HP VMX Device Driver."

These two sets of planes exist with little interaction. For example, when a menu in the overlay planes occludes a window in the image planes, no *expose event* is generated when the menu is removed because the image frame buffer contents are not affected. Only other windows in the image planes generate *expose events* for image plane windows.

Color map index 255 for the overlay planes is the transparent color on the CRX-48Z. Any overlay pixels containing this index will force the image planes to be displayed.

Per-Window Double-Buffering and Color Maps

The CRX-48Z supports double-buffering on a per-window basis. The CRX-48Z supports 8/8 planes double-buffered for each of the Starbase color map modes (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`), and 24/24 planes double buffered for `CMAP_FULL`.

X11 always renders to the currently visible buffer when a window has double-buffering enabled.

Pixels in the image planes are not displayed unless the corresponding pixel in the overlay planes is set to 255.

X11 Cursor and Starbase Echos

On previous devices, Starbase echos were either a hardware cursor, or were software echos rendered in the overlay or image planes. The CRX-48Z device has a hardware cursor (similar to many other devices), but this cursor is always used by the X server, and is not available to Starbase programs. All Starbase echos are rendered in special cursor planes (these planes are available ONLY for Starbase echos). These special cursor planes are separate from the image and overlay planes, and do not interfere with the graphics in those planes.

The following list describes the behavior of Starbase echos on the CRX-48Z device, and decribes some differences from other devices.

- Since the echos are not rendered in the image or overlay planes, they are "combined" with the image plane graphics simply by obscuring them, rather than by an `XOR` operation.
- Although there may be multiple Starbase echos on the CRX-48Z device, there is only a single set of color registers (foreground and background) for those echos. All echos are displayed using those same color registers, so at any given time, all echos on the screen will be the same color. This may cause color "flickering" if multiple echos are defined with different colors. The color will be that of the echo most recently updated (moved).
- Since there are only two color registers, full depth (i.e. 256 colors) Starbase raster echos are not available on the CRX-48Z device. When defining a raster echo (see `define_raster_echo()` in the *Starbase Reference Manual*), all source values of 0 will be treated as background, and all other values will be treated as foreground. (To change the default background value, see the `R_ECHO_FG_BG_COLOR` gescape). A transparency mask may be specified for Starbase echos as per the `R_ECHO_MASK` gescape.
- Non-transparent graphics in the overlay planes (such as overlay plane windows, X applications, etc.) can obscure the Starbase echo (if the image plane window

is obscured). However, transparent sections of overlay windows will NOT obscure the Starbase echo.

- A common operation may be to have a window in the image planes with a transparent child window in the overlay planes. In this case, the Starbase echo should be associated with the image-plane file descriptor, rather than the overlay-plane child window. For device independent behavior in this situation, use the `inquire_capabilities()` Starbase entry point. If the `IC_TRANS_WIN_IMAGE_CURSOR` bit of the `CONTROL_CAPABILITIES` byte is set, put the echo in the image plane window. If this bit is not set, the echo should be rendered using the overlay plane window's file descriptor. For the CRX-48Z device, this bit *will* be set.

Supported X Windows Visuals

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 in the *Starbase Graphics Techniques* manual.

It is possible to allocate an 8-plane pseudocolor visual in either the overlay planes or image planes of this device. For details on how to select one or the other, see the example code in the file `/usr/lib/starbase/demos/SBUTILS/wsutils.c`. The X11 `SERVER_OVERLAY_VISUALS` property indicates whether a given visual is in the overlay planes or not.

How to Read the Supported Visuals Table

The table of Supported Visuals contains information for programmers using either Xlib graphics or Starbase. The table lists what depths of windows and color map access modes are supported for the CRX-48Z. It also indicates whether or not backing store is available for a given visual. The table also lists the double-buffer configurations supported by Starbase for this device driver.

Table 5-2. CRX-48Z Supported Visuals

Device	Depth	Visual Class	Backing Store		Starbase Doublebuffer
			Xlib	Starbase	
CRX-48Z Overlay Planes[1]	8	PseudoColor	Yes	Yes[2]	8/8 [3]
CRX-48Z Image Planes	8 24	PseudoColor DirectColor	Yes Yes [4]	Yes[2] No	8/8 [3] 24/24

[1] There are only 255 overlay colors (0 - 254). Color index 255 is reserved for transparency.

[2] Backing store is supported in depth 8 visuals only via the `hpvmx` driver.

[3] 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.

[4] Xlib backing store supported except for Starbase `gopen`ed windows.

To Set Up the Device

The `mknod` command (see `mknod(1m)` in the *HP-UX Reference* manual) creates a device special file that is used to communicate between the computer and the display device. The name of this device special file should be in the X*screens file to specify to the X Server which device to use. HP-UX will usually automatically create the device file for the primary graphics display on the system. For the CRX-48Z, only the device special file name is needed in the X*screens file.

If the need ever arises to create a device special file for the CRX-48Z, by convention, it should be created in the `/dev` directory. Any name may be used for the device special file, however the name that is suggested for the default device is `/dev/crt`.

To Create Special Device Files (mknod)

For an SPU with only one SGC interface slot, a sample `mknod` entry would be:

```
/etc/mknod /dev/crt c 12 0x100000
```

For an SPU with two SGC interface slots, a sample `mknod` entry for a device in the second slot would be:

```
/etc/mknod /dev/crt1 c 12 0x000000
```

To Compile and Link with the Device Driver

For Shared Libraries

The compiler driver programs (`cc, fc, pc`) link with shared libraries by default. The shared device driver is the file named `libddcrx48z.sl` in the `/usr/lib` directory. Starbase will explicitly load the device driver at run time when you compile and link with the Starbase shared library `/usr/lib/libsb.sl`, or use the `-lsb` option. This loading occurs at `gopen`(3G) time.

The window libraries must be linked with all programs that use the CRX-48Z.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11\
-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

For Archive Libraries

The archive device driver is located in the `/usr/lib` directory with the file name `libddcrx48z.a`.

Note Although the `hpvmx` driver is used to render Starbase Graphics to the CRX-48Z overlay planes, it does *NOT* need to be included in the link sequence.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libddcrx48z.a`
2. an appropriate relative path name
3. the `-lddcrx48z` option with the `LDOPTS` environmental variable exported and set to `"-a archive"`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -lddcrx48z\
-L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -lddcrx48z\
-Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

or for Pascal, use:

```
pc example.p -lddcrx48z\
-Wl,-L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm -o example
```

To Open and Initialize for Output

Syntax Examples

C programs:

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

FORTRAN 77 programs:

```
fildes = gopen('/dev/screen/window'//char(0), OUTDEV,
char(0), INIT)
```

Pascal programs:

```
fildes := gopen('/dev/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`(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 a `NULL` parameter. In this case, Starbase will automatically choose the appropriate driver.

  For example:

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

  A character string may be used to specify the driver. For example:

`"hpcrx48z"`	*for C*
`'hpcrx48z'//char(0)`	*for FORTRAN 77*
`'hpcrx48z'`	*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 more details of `gopen` actions when accessing an X Window.
  - `0` (zero) - Open the window, but do nothing else. The software color table is initialized from the current state of the hardware color map for that window.
  - `INIT` - Open and initialize the device as follows:
    1. The window is cleared to 0s. For overlay windows, only the overlay planes are cleared. For image windows, the 48 image planes are cleared.
    2. The color map is reset to its default values. For depth 8 windows, the color map is initialized as `CMAP_NORMAL`. For depth 24 windows, the color map is initialized as `CMAP_FULL`.
    3. Clear the Z-buffer.
  - `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.

  This mode is provided for compatibility of integer precision with previous devices. `INT_XFORM` will 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`.
- □ `UNACCELERATED` - This flag is ignored.
- □ `ACCELERATED` - This flag is ignored.

A table located in the section "HP VMX: The CRX-family Overlay Plane Driver" in the "HP VMX Device Driver" chapter in this manual lists which driver will be selected at `gopen` based on whether:

- A window is in the image planes or overlay planes.
- The window has backing store or not.
- The `hpcrx48z`, `hpvmx`, or `NULL` driver is specified.

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.

Device Defaults

Dither Default

The number of colors allowed in a dither cell is 1, 2, 4, 8 or 16. The default value is 16. Selecting a color with the `fill_color` procedure will allow dithering for filled areas when desired.

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.

Plane Mask Defaults

All accessible planes are enabled for display and writing.

Semaphore Default

Semaphore operations are enabled.

Line Type Defaults

The default line types are created with the dash lists shown below. The Starbase default line type is `SOLID`, line type 0. See the following table.

Table 5-3. Default Line Types

Line Type	Dash Count	Dash List
0	1	1
1	2	8 8
2	2	1 1
3	4	13 1 1 1
4	6	11 1 1 1 1 1
5	2	11 5
6	4	12 1 2 1
7	6	9 1 2 1 2 1

Each number in the above table represents a single dash in the pattern. The first number is the number of dashes in the list. The rest of the numbers are the length of each dash, alternating on and off, with the first dash being on. For example, line type 6 has 4 dashes, 12 pixels on, then 1 off, then 2 on, then 1 off. This pattern repeats for the length of the line.

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 5-4. 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 following calls are not supported when using the CRX-48Z. When executed, these calls will produce no result (i.e. they are no-ops).

```
bf_texture_index
define_texture
texture_index
texture_viewport
texture_window
```

Conditional Support of Starbase Calls

The following calls are supported with the listed exceptions:

`alpha_transparency`	Alpha only applies to filled areas such as polygons, quadrilateral meshes, triangular strips, and spline surfaces. Alpha per vertex is supported. Vector primitives are not rendered with `alpha_transparency`. The alpha_transparency feature is limited to `CMAP_FULL` in the 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 CRX-48Z does not support alpha transparency with attenuation. (See the Reference page for `alpha_transparency` for parameters in the *Starbase Reference Manual*).
`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 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 displayed (`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` gescape. 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.

`define_raster_echo`

The `hpcrx48z` driver interprets pixel values of 0 in the raster echo definition to be the cursor background color. It interprets all non-zero pixel values to foreground color.

`line_filter`
`perimeter_filter`

The antialiasing features are only available with the `CMAP_FULL` color map mode in the 24-plane configurations.

The CRX-48Z has two antialiasing 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 three pixel wide output. Pixel values are multiplied by the alpha value and blended with the background according to the formula:

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

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

$pixel\_color = (new\_pixel \times \alpha) + old\_pixel;$

`pattern_define`	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.
`screenpr`(1)	Because of per-window double-buffering and multiple color maps, this utility is only supported for printing one window at a time. Attempts to print pixels outside this one window may result in a wrong color or pixels from the wrong buffer being printed.
`shade_mode`	Shading is not supported on device coordinate primitives.
`text_precision`	Only `STROKE_TEXT` precision is supported.

Supported Gescapes

The hpcrx48z driver supports the following gescape operations. Refer to Appendix A of this manual for details on gescapes.

- BAD_SAMPLE_ON_DIFF_SCREEN—Restore the locator and choice sampling of the X11 pointer device.
- BLOCK_WRITE_SKIPCOUNT—Specify byte skip count during block write.
- CONTOUR_CONTROL—Specify alternative methods for interpolation of contour data.
- 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.
- GR2D_PLANE_MASK—Overrides the bit mode mask.
- 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.
- OLD_SAMPLE_ON_DIFF_SCREEN—Inquire the locator and choice sampling of the X11 pointer device.
- PLUG_ACCELERATED_PIPELINE—Controls the rendering of the graphics accelerators into the frame buffer.
- POLYGON_TRANSPARENCY—Segment control over front/back face “screen door”.
- R_BIT_MASK—Select plane for reading and writing bit blocks.
- R_BIT_MODE—Specify data format for bit/pixel block transfer operations..
- R_DEF_ECHO_TRANS—Define transparency mask for raster echo.
- R_ECHO_FG_BG_COLORS—Define color attributes.
- R_ECHO_MASK—Turns on echo transparency mask.
- R_GET_FRAME_BUFFER—Read the frame buffer and control space addresses.
- R_LINE_TYPE—User defined line style and repeat length.
- R_LOCK_DEVICE—Lock device.
- R_UNLOCK_DEVICE—Unlock device.
- READ_COLOR_MAP—Read color map.
- 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.
- `ZBANK_ACCESS`—Enable/disable the Z-buffer bank for reading and writing.
- `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 - Enable/disable gamma correction. Gamma correction is implemented differently on the `hpcrx48z` than on older devices. It is implemented in the color map rather than in the frame buffer. For information on the gescape `GAMMA_CORRECTION`, refer to Appendix A.
- R_ECHO_CONTROL - Regardless of what cursor is requested, hardware or software, the CRX-48Z will always return the arg2[0] value of 1 which indicates the hardware cursor is being used.

FINAL TRIM SIZE : 7.5 in x 9.0 in

6

The HP Entry Level VRX Device Driver

Device Description

The HP Entry Level VRX (EVRX) device driver supports a family of HP workstation products with built-in graphics hardware. The internal graphics hardware consists of an 8 image plane dumb frame buffer which is tightly coupled to the host SPU via the SGC bus. This architecture yields competitive performance and low cost due to the simplified graphics hardware design.

The HP Entry Level VRX is supported on the HP 425E workstation in the following configurations:

- 1280x1024 color graphics option
- 1280x1024 gray scale graphics option
- 1024x768 color graphics option

The HP Entry Level VRX is also supported on the HP 382 computer system in the following configurations:

- 1024x768 color graphics
- 640x480 color graphics
- 640x480 grayscale graphics

The 8 image planes are organized as an array of bytes, with each byte representing a pixel (a bit-mapped display). To produce a color on the screen, each 8-bit pixel value is treated as an index into a 256 entry color table. This allows the simultaneous display of 256 individual colors from a palette of 16 million colors. On the grayscale configurations, 256 shades of gray can simultaneously be displayed.

Overview of Device Capabilities

The HP Entry Level VRX device driver supports 2D and wireframe 3D graphics. It does not support more advanced features such as lighting, shading, depth cueing, texture mapping, or Z buffering. A detailed list of unsupported Starbase calls is included at the end of this chapter.

The HP Entry Level VRX device driver supports raster text calls from the fast alpha and font manager libraries. See the *Fast Alpha/Font Manager Programmer's Manual* for more information.

In general, the HP Entry Level VRX device driver is very similar in capabilities to the HP 98550 device driver.

Setting Up the Special Device Files (mknod)

The special device file for the internal Entry Level VRX graphics hardware should already be setup correctly as `/dev/crt` by the system boot code. If `/dev/crt` does not exist, is deleted for some reason, or does not allow the device driver access to the hardware, use the information below to create a new `/dev/crt`.

The `mknod(1M)` command creates a special device file which is used to communicate between the SPU and the internal graphics. Superuser capability (the root login) is required to setup special device files. See the *HP-UX Reference* manual for more information on the `mknod(1M)` command.

The correct `mknod` command syntax to create an SGC bus special device file for the internal Entry Level VRX graphics hardware is:

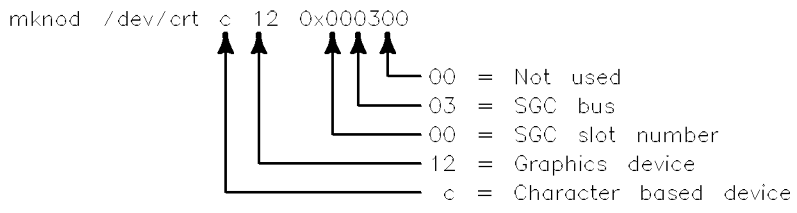

Figure 6-1.

Note When using the `mknod` command to create a special device file for the HP 382, the correct minor number is `0x840300` instead of `0x000300`.

Linking the Driver

The HP Entry Level VRX device driver is located in the `/usr/lib` directory with the file name `libddevrx.a`. The shared library version of the driver is named `libddevrx.sl` This device driver can be directly linked with the application program (archive option) or dynamically loaded at run-time (shared option). The shared library option reduces executable disk space requirements at the expense of a slight performance penalty.

Shared Library Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11\
-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

The dynamic loader is -ldld. The math library, -lm, is needed for some applications.

For Archive Libraries

The archive device driver is located in the `/usr/lib` directory with the file name `libddevrx.a`.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libddevrx.a`
2. an appropriate relative path name
3. the `-lddevrx` option with the `LDOPTS` environmental variable exported and set to `"-a archive"`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -lddevrx -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -lddevrx -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -lddevrx -Wl,-L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Initialization

The gopen procedure has four parameters: Path, Kind, Driver, and Mode.

Path	This is the name of the special device file created with the `mknod` command, e.g. `/dev/crt`.
Kind	This parameter must be `OUTDEV`, unless used for an X11 window, in which case `OUTINDEV` may be used.
Driver	The character representation of the driver type, namely:

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

This parameter can be `NULL` or `0`, in which case, the `gopen` command will automatically determine the correct driver. This feature is highly recommended for device independence.

Mode	The mode consists of several flag bits or'ed together. The most common mode is `INIT`. See the *Starbase Reference* manual for complete details on the `gopen` mode bits.

Syntax Examples

Examples to open and initialize the HP Entry Level VRX driver for output:

C Programs:

Explicit device driver:

```
fildes = gopen("/dev/crt",OUTDEV,"hpevrx",INIT);
```

Automatically determine the device driver:

```
fildes = gopen("/dev/crt",OUTDEV,NULL,INIT);
```

FORTRAN 77 Programs:

Explicit device driver:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,'hpevrx'//char(0),INIT)
```

Automatically determine the device driver:

```
            fildes = gopen('/dev/crt'//char(0),OUTDEV,char(0),INIT)
```

Pascal Programs:

Explicit device driver:

```
            fildes := gopen('/dev/crt',OUTDEV,'hpevrx',INIT);
```

Automatically determine the device driver:

```
            fildes := gopen('/dev/crt',OUTDEV,0,INIT);
```

Porting Guide

Differences from other Starbase Device Drivers

- The HP Entry Level VRX graphics driver does not support the `SB_DISPLAY_ADDR` environment variable. The HP-UX kernel will always pick the best virtual address to map in the graphics hardware.
- ALL rendering operations are performed correctly to the screen as well as backing store (retained raster) for applications running in an X11 window.
- There is no physical "offscreen" memory on the Entry Level VRX graphics device. Applications which either use `R_FULL_FRAME` buffer or try to write to offscreen memory will not work.
- Application which use the `gescapes GR2D_FILL_PATTERN` or `R_DEF_FILL_PAT` should use the Starbase call `pattern_define` to define fill patterns.
- Gray scale configurations will run color applications by approximating the colors with the closest shade of gray. Application tuning for gray scale is encouraged, but not necessary.
- It is not necessary to link in the `libddbyte.a` and `libddbit.a` libraries into the application for Starbase to work correctly in an X11 window with backing store enabled (retained raster).

Gray Scale Configurations

The software to handle gray scale actually resides in the higher level Starbase code, not the driver. It has been designed so that color applications can run without modification using the closest matching shades of gray. Application optimization for gray scale is encouraged, but not necessary.

All the Starbase color map modes: `CMAP_NORMAL`, `CMAP_FULL`, and `CMAP_MONOTONIC` are supported on the gray scale configurations of the Entry Level VRX graphics hardware.

Conceptually, when the color map is initialized or redefined, it is treated in the same way as if the device were color. Then, as a last step, the color map is put through one more transformation to convert the color red, green, and blue values into an equivalent intensity of gray. The formula used to convert from color to gray scale is:

```
Intensity = (0.30 * Red) + (0.59 * Green) + (0.11 * Blue)
```

Buffering

The HP Entry Level VRX driver is a buffered driver. A buffered driver queues graphics requests in a memory buffer, rather than executing them immediately. When the buffer is full, or, when the application calls the Starbase `make_picture_current` procedure, the buffer is traversed and the graphics requests executed. This is referred to as a buffer flush.

Buffering can be a performance enhancing feature because the driver can perform some of the overhead associated with rendering once, rather than for each individual graphics request.

The size of the buffer can affect application performance. Too large of a buffer can reduce interactivity. Too small of a buffer negates the overhead advantages of buffering and can reduce performance of an application that typically requests large amounts of data to be rendered at once - such as a 1000 vector polyline requests.

See *Device Specific Performance Tips* in this chapter for a way to tune the HP Entry Level VRX driver's buffer size for your application.

Device Coordinate Addressing

For device coordinate operations, location (0,0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The coordinates of the lower-right corner of the screen can be obtained by taking the width and height of your display (dependent on the particular configuration) and subtracting one from these numbers. For example, the lower-right corner of the screen for the 1280 by 1024 resolution is location (1279,1023).

Direct Frame Buffer Access

If you have the base address of the frame buffer from the `gescape` `R_GET_FRAME_BUFFER`, you can access pixel `(X,Y)` by adding the offset `(X + Y * 2048)` to the base address and performing a byte access. Four consecutive pixels can be accessed by doing a long word (32-bit) access. Even though each scanline takes up 2048 bytes worth of address space, there is no offscreen memory. Writing to non-existent offscreen memory may cause you application to crash.

If you are directly accessing the frame buffer please use the `R_LOCK_DEVICE` `gescape` just prior to direct frame buffer access and `R_UNLOCK_DEVICE` immediately after you are done accessing the frame buffer.

Device Specific Performance Tips

1. The gescape SET_BUFFER_SIZE can be used to dynamically tune the buffer size inside the application. The buffer can range from a minimal size of 512 bytes (the default) to a maximum size of 4096 bytes. Values outside this range will be clipped.
2. Since the hardware does not have support for replacement rules (drawing modes), source replacement rule is much faster than non-trivial replacement rules, such as exclusive-or.
3. With a fast SPU and a high bandwidth bus to the frame buffer, `block_write` and `block_read` will perform significantly faster than graphics devices which utilized DIO and DIO-II as their path to the frame buffer.
4. Rendering to an X11 window will be significantly slower if that window has backing store. However, if the graphics in that window are complex enough, backing store could be a performance win when it comes to refreshing the regions of the window that were obscured.
5. If you are using the Starbase device coordinate entrypoints to do your rendering (e.g. `dcpolyline2d`), you can increase performance slightly by opening the device with the mode flags INIT or'ed with INT_XFORM, rather that merely opening with INIT.
6. The Entry Level VRX graphics hardware does have some support for bit/pixel frame buffer access. This hardware is used when the R_BIT_MODE gescape is used and `block_write` is called with the raw flag `TRUE`.

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 6-1. HP Entry-Level VRX (EVRX)

Model 425E Integrated Graphics	High-Res Color
Model 425E Integrated Graphics	Medium-Res Color

The supported server mode is Image mode.

Table 6-2. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	doublebuffer?	Xlib	Xlib	SGL	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	

Table 6-3. HP Entry-Level VRX (EVRX)

Model 425E Integrated Graphics	High-Res Grayscale

The supported server mode is Image mode.

Table 6-4. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	doublebuffer?	Xlib	Xlib	SGL	
8	No	Grayscale	●	●	
	Yes (4/4)	Grayscale	●	●	

X11 Cursors and Starbase Echos

The following list shows default positions where the Starbase echo and X11 cursor (called echo and cursor, respectively) reside for each of the X11 server operating modes.

HP Entry-Level VRX (EVRX)

- Image Mode

  X11 cursor uses hardware cursor. Starbase echos are in the image planes.

Starbase Unsupported Commands

The following commands are not supported for use with this driver. Calls to these procedures will have no effect:

```
alpha_transparency             depth_cue_range
backface_control               depth_cue-range
bank_switch                    hidden_surface
bf_alpha_transparency          interior_style (INT_OUTLINE)
bf_control                     interior_style (INT_POINT)
bf_fill_color                  light_ambient
bf_interior_style              light_attenuation
bf_perimeter_color             light_model
bf_perimeter_repeat_length     light_source
bf_perimeter_type              light_switch
bf_surface_coefficients        line_filter
bf_surface_model               perimeter_filter
bf_texture_index               set_capping_planes
contour_enable                 set_model_clip_indicator
define_contour_table           set_model_clip_volume
define_texture                 shade_range
define_trimming_curve          surface_coefficients
deformation_mode               surface_model
depth_cue                      texture_index
depth_cue_color                texture_viewport
                               texture_window
                               viewpoint
                               zbuffer_switch
```

Starbase Conditionally Supported Commands

`block_read,block_write`	The `raw` parameter is normally ignored unless the following `gescapes` have been used: `BLOCK_WRITE_SKIPCOUNT` or `R_BIT_MODE`.
`shade_mode`	The color map mode may be selected but shading can not be turned on.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	The ⟨*use*⟩ parameter must be zero, any extra coordinate information will be ignored.
`with_data`	`partial_polygon_with_data3d` `polygon_with_data3d` `polyhedron_with_data` `polyline_with_data3d` `polymarker_with_data3d` `quadrilateral_mesh_with_data` `triangle_strip_with-data` Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

Gescapes

The following `gescape` functions are supported by the HP Entry Level VRX device driver. Detailed information about these functions can be found in appendix A of this manual.

- `SWITCH_SEMAPHORE`—Enable or disable locking.
- `READ_COLOR_MAP`—Read hardware color map.
- `IGNORE_RELEASE`—X11: Events triggered on button press only.
- `TRIGGER_ON_RELEASE`—X11: Events triggered on button press and release.
- `R_BIT_MASK`—Set plane mask for bit/pixel block reads/writes.
- `R_BIT_MODE`—Enable bit/pixel block operations when raw flag TRUE.
- `R_GET_FRAME_BUFFER`—Get device control space and framebuffer addresses.
- `R_LOCK_DEVICE`—Lock the device..
- `R_UNLOCK_DEVICE`—Unlock the device.
- `R_LINE_TYPE`—Define a 16-bit line pattern.
- `BLOCK_WRITE_SKIPCOUNT`—For byte/pixel block writes when raw flag TRUE.
- `SET_BUFFER_SIZE`—Dynamically tune driver buffer size.

7

The PersonalVRX Device Driver

Device Description

The PersonalVRX Graphics Display Station is a graphics subsystem which interfaces with the host SPU, utilizing a high-resolution (1280×1024) color display (purchased separately). See the *Introduction* section of this manual for systems supporting these controllers.

The PersonalVRX Graphics Display Station is available in the following configurations:

- HP 98705A and HP 98705C
  - 8 image planes (frame buffer).
- HP 98705B
  - 16 image planes (frame buffer).
  - 16 planes Z-buffer.

Two device drivers are provided to access the PersonalVRX Display:

- `hp98704`—The HP 98704 Device Driver is used to access the graphics display, without using the graphics accelerator, in raw mode or X11 windows.
- `hp98705`—The HP 98705 Device Driver is used to access the graphics display, utilizing the graphics accelerator, in raw mode or X11 windows.

This section covers the `hp98704` and `hp98705` device drivers.

The display produces a resolution of 1280x1024 pixels. The standard color display system (HP 98705A or HP 98705C) has eight planes of frame buffer to provide 256 simultaneous colors, plus four overlay planes for non-destructive alpha, cursors, or graphics. A fully configured system (HP 98705B) includes two banks of frame buffer memory, four overlay planes, and a dedicated board for full 16 bit Z-buffer capability.

An eight-plane configuration allows 256 colors to be displayed simultaneously from a palette of 16 million. A 16-plane configuration may be treated as two

8-plane frame buffers where only one buffer is displayed at a given moment in time. This configuration is useful for double buffering. The 8-plane system can also be configured to display 3 bits red, 3 bits green and 2 bits blue per pixel in `CMAP_FULL` mode. Double buffering can be done in `CMAP_FULL` mode with the 16-plane configuration.

The display system is a memory-mapped device with special hardware for:

- Write enable/disable individual planes.
- Video enable/disable individual planes.
- Frame buffer writes with specified replacement rule (see drawing_mode).
- Arbitrary sized rectangular frame buffer to frame buffer copies.
- Bit per pixel block reads and writes.
- Raster and crosshair cursors.
- Video blinking of individual planes.
- Video blinking of individual color map locations.

The accelerated `hp98705` device driver also features:

- Write enable/disable of pixels in a 4x4 cell for "screen door" transparency.
- An Intel i860 processor with hardware floating-point processing for high-speed 3D operations.
- NMOS III scan converter with six axis interpolation for Gouraud-shaded, Z-buffered vectors and polygons.
- Pixel clipping for full speed graphics to obscured windows.
- Dedicated 2K by 1K 16-bit Z-buffer (HP 98705B only).

The display is organized as an array of bytes, with each byte representing a pixel on the display. Since there are eight planes of frame buffer memory, color map indices range from 0-255. The color map is a RAM table that has 256 addressable locations and is 24 bits wide (8 bits each for red, green, and blue). Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

In addition to the frame buffer banks of eight planes each, four overlay planes are provided. These overlay planes have their own unique color map, separate from the color map used for the image planes. This overlay color map consists of sixteen 24-bit entries, allowing the user to select 16 colors from the full palette of over 16 million choices In addition, each entry in the overlay color map may be made **dominant** (opaque) or **non-dominant** (transparent). A **dominant** entry causes all pixels in the overlays set to that value to display the color in the overlay map, regardless of values in the image planes "below".

A **non-dominant** (or transparent) entry causes pixels with that value to display the color in the image planes "below". By default, the `hp98705` device driver sets all overlay color map entries to be **dominant** when opened to the overlays. Entries may be set to be **non-dominant** with the Starbase `gescape, R_TRANSPARENCY_INDEX`.

You can use overlay planes for non-destructive alpha, graphics, or cursors. For example, when the HP 98705 is used as system console, the Internal Terminal Emulator (ITE) uses three of the overlay planes for alpha information. By doing so, there is no interaction between ITE text and images in the graphics planes. To produce graphical images in the overlay planes, the `hp98704` or `hp98705` device driver may be opened directly to the overlay planes, as if they were a separate device. (Refer to the section **Setting up the Device** for more information.)

Typically, an application does not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase does provide the `gescape` function `R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer (this `gescape` is discussed in the appendix of this manual). Frame buffer locations are subsequently accessed based upon their offset from this returned address. The first byte of the frame buffer (byte 0) represents the upper left corner pixel of the screen. Byte 1 is immediately to its right, Byte 1279 is the right-most pixel on the top scan line. The next 768 bytes of the frame buffer are not displayable. Byte 2048 is the left-most pixel on the second scan line from the top. The lowest right corner pixel on the screen is byte number 2,096,383. If more than one frame buffer bank is installed then bank switching must be used to access the additional memory. A number of Starbase calls may set the bank register so it is advisable to call `bank_switch` just prior to making accesses to the frame buffer pointer to ensure the desired results.

The off-screen portion of the frame buffer may be accessed via the `gescape` procedure `R_FULL_FRAME_BUFFER`, documented in the appendix of this manual. Care should be exercised with this `gescape`, as Starbase, X Windows, and other processes make use of the frame buffer off-screen memory.

High-Performance Bit-per-pixel Support

The PersonalVRX provides hardware support for high speed bit-per-pixel block reads and writes. Bit-per-pixel mode is set by using the gescape, `R_BIT_MODE`. When in this mode, one byte of data represents data for eight pixels. See the description of the `R_BIT_MODE gescape`, located in the appendix of this manual, for more details.

Multiple-plane bit-per-pixel Support

The `gescape`, `GR2D_PLANE_MASK`, defines a mask that allows multiple planes to be read or written.

The definition of `GR2D_PLANE_MASK` requires data array space for each plane that will be read or written, and each is done individually For example, if the mask is `01101` then the user's data array must look like:

Plane 0 Data
Plane 2 Data
Plane 3 Data

See the `gescape`, `GR2D_PLANE_MASK`, for more details.

Bit-per-Pixel Replacement Rule per Plane

The PersonalVRX supports a replacement rule per plane while doing bit-per-pixel block writes. This allows a replacement rule to be set individually for each plane. These replacement rules could be used, typically, to provide fast two color bit-per-pixel block writes. See the description of the `gescape`, `SET_REPLACEMENT_RULE`, for more details. Performing block writes using the per/plane replacement rule may work twice as fast in the overlay planes as it does in the image planes, depending on the rules used.

Setting Up the Device On Series 300 and 400

The `hp98704` and `hp98705` device drivers can be used with the graphics display configured only in DIO-II address space. Refer to the **Configuration Reference Manual** for a description of DIO-II address space.

Note If the PersonalVRX is configured as an external display, then there will not be an Internal Terminal Emulator (ITE) for that device. Since it is the ITE that normally initializes the display, external devices must be reset after power-up by running a simple Starbase program with a mode of `RESET_DEVICE` in the `gopen` call. It may also be necessary to run this program after running an application which manipulated the overlay color map, such as a windows application program. An example program which could be called from /etc/rc during power-up is given at the end of this section. For more details concerning the effects of `RESET_DEVICE`, see the **Device Initialization** information in this section.

The Graphics Interface card may be installed in any DIO II slot in the computer's backplane or in any I/O slot of the expander.

DIO-II Switch Settings

The graphics interface card has a single 5-bit address select switch. The switches are labeled (left to right) from 0 to 4. The leftmost switches represent the most significant bits, hence, switch 0 is the most significant bit of the address, and switch 4 is the least significant. Select the address space by setting the switches either to open or not open. Switches in the open position represent 0's while switches not in the open position represent 1's. See the address table below for the switch-setting/address-mapping relationship.

The PersonalVRX can only be used in DIO-II address space. In this mode, the five switches determine the DIO-II select codes to be used. A PersonalVRX will use three DIO-II select codes. Both the frame buffer and control space reside in the select code areas.

The control space requires 4 Mbytes of space, starting at `CTL_BASE`. The five switches described above determine the address of `CTL_BASE`. The frame buffer requires 8 Mbytes of space, starting at `FB_BASE`.

PersonalVRX DIO-II Control Space Settings

Switch Setting MSB to LSB	CTL_BASE	DIO-II Select Code	FB_BASE
00001	$01400000	133	$01800000
00010	$02400000	137	$02800000
00011	$03400000	141	$03800000
00100	$04400000	145	$04800000
00101	$05400000	149	$05800000
00110	$06400000	153	$06800000
00111	$07400000	157	$07800000
01000	$08400000	161	$08800000
01001	$09400000	165	$09800000
01010	$0A400000	169	$0A800000
01011	$0B400000	173	$0B800000
01100	$0C400000	177	$0C800000
01101	$0D400000	181	$0D800000
01110	$0E400000	185	$0E800000
01111	$0F400000	189	$0F800000
10000	$10400000	193	$10800000
10001	$11400000	197	$11800000
10010	$12400000	201	$12800000
10011	$13400000	205	$13800000
10100	$14400000	209	$14800000
10101	$15400000	213	$15800000
10110	$16400000	217	$16800000
10111	$17400000	221	$17800000
11000	$18400000	225	$18800000
11001	$19400000	229	$19800000
11010	$1A400000	233	$1A800000
11011	$1B400000	237	$1B800000
11100	$1C400000	241	$1C800000
11101	$1D400000	245	$1D800000
11110	$1E400000	249	$1E800000
11111	$1F400000	253	$1F800000

A DIO-II display may be used as the system console or as an external display. In order to use the display as the system console, it must be configured as the first DIO-II display in the system, and there must be no DIO-I console, or remote terminals. Being the first DIO-II device means that it has the lowest DIO-II select code in the system. In order to use a PersonalVRX as a DIO-II system console, select code `133` is recommended.

Note It is necessary to increase some of the HP-UX tunable system parameters due to the size of the DIO-II mapping of a PersonalVRX. For details on how to reconfigure your kernel, refer to the **HP-UX System Administrators Manual** (particularly the "Configuring HP-UX" section in "The System Administrators Toolbox" and the "System Parameters" appendix.

It is essential that you consult the above referenced HP-UX documentation before you attempt to reconfigure your system. It is possible to adversely affect your HP-UX system if a mistake is made. Ensure you have an understanding of these procedures before proceeding.

Example Program to Reset the PersonalVRX

The following example uses the hp98705 device driver. The hp98704 device driver can be substituted.

```
/*
 * Starbase program: reset98705.c
 * Compile: cc -o reset98705 reset98705.c -ldd98705 -lsb1 -lsb2 -lm
 * Destination: /usr/bin
 * Execute: add line to the /etc/rc -
 *   "/usr/bin/reset98705 /dev/crt"
 *
 * Example program to be put in /etc/rc for resetting
 * an external PersonalVRX device during power-up.
 */
 #include <starbase>

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

    if ((fildes = gopen(argv[1],OUTDEV,"hp98705",INIT|RESET_DEVICE))<0)
      printf("External PersonalVRX %s initialization failed.\\n",argv[1]);
    else {
      printf("External PersonalVRX %s initialization succeeded.\\n",argv[1]);
      gclose(fildes);
    }
 }
```

Address Space Usage

The PersonalVRX is memory mapped into a processes virtual address space, starting at the value specified by the environment variable SB_DISPLAY_ADDR. If this variable is not set, then mapping defaults to 0xB00000. The control space starts at this address and grows towards larger address values. After the control space comes the frame buffer, then shared memory mapped for Starbase drivers. In DIO-II, control space is 4 Mbytes and the frame buffer is 8 Mbytes. The size of the hp98704/hp98705 drivers' shared memory is slightly less than 300 Kbytes.

If your application maps memory pages to specific addresses, or needs a large stack, then you may need to adjust SB_DISPLAY_ADDR to avoid conflicts.

Special Device Files (`mknod`)

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod(1M)` information in the **HP-UX Reference** for further information. The name of this special device file is passed to Starbase in the gopen procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however the name that is suggested for these devices is `crt`.

The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the `mknod` command.

Since the device is in DIO-II address space, (refer to the "Switch settings" section) the `mknod` parameters should create a character special device with a major number of `12` and a minor number of `0xSc0200` where `Sc` is the external select code in hexadecimal notation.

```
mknod /dev/crt  c  12  0xSc0200
```

The PersonalVRX may also be used for the overlay planes in graphics mode. The minor number may be set to cause Starbase drivers to use either three or four overlay planes. When running to three planes, one plane is still reserved for cursors. When running to all four overlays, only the hardware cursor is available for Starbase graphics echoes. If more than one echo is requested, or if another process is using the cursor, then the request for another echo will fail. Note that since the terminal emulator and window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is opened in this manner while these processes are present. To open the PersonalVRX to three overlay planes instead of the graphics planes, the last byte of the minor number must be one. To run to all four overlays, the last byte of the minor number must be three.

Thus, to create a device file that will allow the PersonalVRX to use only three overlay planes, the following command should be used:

```
mknod /dev/ocrt  c  12  0xSc0201
```

where SC is the external select code in hexadecimal notation.

To create a device file that uses all four overlay planes, the following command should be used:

```
mknod /dev/o4crt c 12 0xSc0203
```

Series 700 System Configuration

Following are some examples of using the mknod entry for the HP-UX Operating System.

For an SPU with only one SGC interface slot, a sample mknod entry would be:

```
/etc/mknod /dev/crt c 12 0x100000
```

For an SPU with two SGC interface slots, a sample mknod entry for the other slot would be:

```
/etc/mknod /dev/crt c 12 0x000000
```

To Compile and Link with the Device Driver

For Shared Libraries

The shared `hp98704` device driver is the file named `libdd98704.sl` in the `/usr/lib// directory. The shared hp98705` device driver is the file named `libdd98705.sl` in the `/usr/lib` directory.

The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
   -lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
   F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
   -lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
   pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
   -lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Archive Libraries

The archive `hp98704` device driver is the file named `libdd98704.a` in the `/usr/lib` directory. The archive `hp98705` device driver is the file named `libdd98705.a` in the `/usr/lib` directory.

As an example, the `hp98705` device driver may be linked to a program using the absolute path name `/usr/lib/libdd98705.a`, an appropriate relative path name, or by using the `-l` option as in `-ldd98705` with the `LDOPTS` environmental variable set to `-a archive`.

The reason for using the `LDOPTS` environmental variable is that the `-l` option will look for a shared library driver first and then look for the archive driver if shared was not found. By exporting the `LDOPTS` variable as specified above, the `-l` option will only look for archive drivers. For more information, refer to the *Programming on HP-UX* manual on linking shared or archive libraries.

This driver also requires the math library to be linked with C programs.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
   export LDOPTS="-a archive"
```

and then:

```
   cc example.c -ldd98705 -L/usr/lib/X11R5 -lXwindow\
   -lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
   F77 example.f -ldd98705 -Wl,-L/usr/lib/X11R5 -lXwindow\
    -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
   pc example.p -ldd98705 -Wl,-L/usr/lib/X11R5 -lXwindow\
    -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 7-1. PersonalVRX Display Types

PersonalVRX [P1]	HP 98705C	High-Res Color
PersonalVRX [P2]	HP 98705A	High-Res Color
PersonalVRX [P3]	HP 98705B	High-Res Color

The supported server modes are Combined and Overlay.

Table 7-2. Windows in Overlay Planes

Contents of X0screens		**Visual Class**	**Backing Store**		**Comments**
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
3	No	PseudoColor	●	●	one color reserved for transparency
4	No	PseudoColor	●	●	one color reserved for transparency

The supported server modes are Combined and Image.

Table 7-3. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	
16	No				Not supported
	Yes (8/8)	PseudoColor	●		only on P3 model

X11 Cursors and Starbase Echos

The following list shows default positions where the Starbase echo and X11 cursor (called echo and cursor, respectively) reside for each of the X11 server operating modes.

PersonalVRX Display, 98704 Device Driver

- Overlay Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

  If image planes are opened and X11 uses three overlay planes, vector echo resides in cursor plane.

  If image planes are opened and X11 uses four overlay planes, vector echo redies in image planes.

  X11 cursor uses hardware cursor.
- Image Mode

  If image-plane X11 window is opened, raster echo resides in image planes and vector echo resides in cursor plane.
- Stacked Screen Mode

  Not supported.
- Combined Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

If image-plane X11 window is opened, raster echo resides in image planes.

If image-plane X11 window is opened and X11 uses three overlay planes, vector echo resides in cursor plane.

If image-plane X11 window is opened and X11 uses four overlay planes, vector echo shares overlay planes.

X11 cursor uses hardware cursor.

PersonalVRX Display, 98705 Device Driver

- Overlay Mode

  The 98705 driver is not supported to overlay planes.

  X11 cursor uses hardware cursor.
- Image Mode

  If image-plane X11 window is opened, echoes reside in the cursor plane and X11 cursor uses hardware cursor.
- Stacked Screen Mode

  Not supported.
- Combined Mode

  If an image-plane X11 window is opened and X11 uses three overlay planes:
  - Vector echoes reside in the overlay planes.
  - Raster echoes are not supported.

  X11 cursor uses hardware cursor.

Usage and Restrictions

hp98704

In no case does the `hp98704` driver support:

- Z-buffering.
- Shading.
- The transform engine.

Use the `hp98705` driver to access these features.

When a raw `gopen` of the overlay planes is done, the `hp98704` driver does not support:

- Bank switching.
- Double-buffering when using three planes. (Double buffering is supported when using four planes.)

When a gopen is done of an X window in the overlay planes, the driver does not support:

- Bank switching.
- Double-buffering.

hp98705

When a device file for the overlay planes is specified at the `gopen` call, bank switching, shading, and depth queuing are not supported.

Up to sixteen unique `gopen` calls, using the `hp98705` driver, may be executed on the same device simultaneously, from any combination of one or more processes.

hp98704 Transparency

By default, the `hp98704` device driver sets all overlay color map entries to be **dominant** when opened to a raw device, except for the entry specified by environment variable `SB_OV_SEE_THRU_INDEX`, which is set to be transparent. This applies to the overlay planes with `INIT` or `RESET_DEVICE`, and to the image planes when `RESET_DEVICE` is used. The default value for `SB_OV_SEE_THRU_INDEX` is 3, however, it can be set to -1 to prevent any entry from being made transparent during initialization. Entries may also be made non-dominant after initialization with the Starbase `gescape`, `R_TRANSPARENCY_INDEX`.

In X windows *Combined* mode, the X server reserves the highest color map entry for transparency (7 or 15, depending on whether three or four overlay planes have been opened). When an X window is opened by a Starbase program, the environment variable is ignored, and the driver uses the same transparency index that the X server is using.

In X windows *Overlay* mode, the X server does not set a default a transparent color. The `hp98704` driver, in an *Overlay* mode window, sets all entries **dominant** when the `gopen` call is executed.

In raw and in X windows *Overlay* mode, an explicit call to `define_color_table`, or use of the `INIT` flag in a call to `shade_mode` or `double_buffer`, will cause transparent entries to be set back to **dominant**. When running in *Combined* mode, the server's transparent entry cannot be made **dominant**.

Cursors

The `hp98704` and `hp98705` device drivers implement cursors using a combination of the hardware cursor and software cursors. If no processes have opened all four overlay planes, then the fourth overlay plane is used for software cursors.

The hardware cursor always appears "over" data in any of the frame buffer planes.

You can think of the fourth overlay plane used for cursors as a separate "cursor plane". Any data in the cursor plane will be displayed "over" data in the graphics planes. Data in the other three overlay planes will be displayed "over" data in the graphics planes and the cursor plane. For example, suppose a graphics application is running in the graphics planes while the window manager is running in three of the overlay planes. If the application has a Starbase cursor in the overlay cursor plane, then the cursor will always be visible inside regions of see-thru because the cursor has display priority over the graphics. If the cursor is moved outside of regions of see-thru then it is not visible since the non-see-thru regions in the overlay planes have display priority over the cursor plane.

The PersonalVRX supports a hardware cursor that is sufficient for all Starbase echo types except rubber-band lines and rectangles. (Rubber-band cursors are supported in software.) Only one hardware cursor is available. Usage of the hardware cursor is defined as follows:

1. If an application is running in a Starbase environment only (ie. X windows are not running), then the hardware cursor is given to the first process that attempts to use a cursor type appropriate to the hardware.
2. By default, if X windows are running, then the window system gets usage of the hardware cursor.
3. The user can control the usage of the hardware cursor via the `gescape` opcode `R_ECHO_CONTROL`. (Refer to the `gescape` appendix in this manual.)

If the hardware cursor is already being used by another process, then overlayed software cursors are used by the `hp98704` driver for vector cursors, and the opened planes are used for raster cursors. If the fourth overlay plane is not available for cursors, then the opened planes are used for all cursor types. Note that the hardware cursor can only support two-color raster cursors; if full-color cursors are needed, the `R_ECHO_CONTROL` `gescape` must be employed to enforce the use of the opened planes for raster cursors.

The `hp98705` driver uses the overlayed software cursors if the hardware cursor is already being used by another process. If the fourth overlay plane is not available for cursors, then an error will be generated when any attempts are made to turn on the cursors.

If a process is using the hardware cursor and it switches to rubber-band cursors, it retains control of the hardware cursor, but the cursor is drawn in the frame buffer (either the fourth overlay plane or the opened planes) using software routines. When the process switches to an appropriate type of cursor, it will again use the hardware cursor. If a user's application never uses hardware-type cursors, then the driver will never attempt to allocate the hardware cursor. However, once the driver has allocated the hardware cursor, the driver does not relinquish control of the hardware cursor until `gclose` is executed. While it is not being used, it simply remains inactive, but no other process can use the hardware cursor once it has been assigned to a process.

If allocation of the hardware cursor is not successful, or the `gescape` is used to force software cursors, then resources for the software cursors are allocated (eg., offscreen areas for raster echo definitions).

The following functions will cause the driver to attempt to allocate cursor resources (that is, either the hardware cursor or software cursor resources):

- `echo_type` or `define_raster_echo`.

- Any of the `gescapes`:
  - `R_DEF_ECHO_TRANS`
  - `R_ECHO_MASK`
  - `R_ECHO_FG_BG_COLORS`
  - `R_OV_ECHO_COLORS`
  - `R_OVERLAY_ECHO`

Device Initialization

Parameters for `gopen`:

- **Path** - This is the name of the special device file created by the **mknod** command as specified in the device setup section (for example, `/dev/crt`).
- **Kind** - This indicates the I/O characteristics of the device, which may be one of the following types:

`INDEV`	Input only.
`OUTDEV`	Output only
`OUTINDEV`	Input and Output.

  Input mode is only possible when the driver is opened to an X window.

- **Driver** - The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire the device and by default load the accelerated driver (if applicable). For example:

`NULL`	for C
`char(0)`	for FORTRAN77
"	for Pascal

  Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is ignored. For example, for the `hp98705` device driver use:

`"hp98705"`	for C
`'hp98705'//char(0)`	for FORTRAN77
`'hp98705'`	for Pascal

- Mode - The mode control word consists of several flag bits ORed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure.
  - □ `SPOOLED` - Cannot be used on raster devices, therefore this flag has no effect with this driver.
  - □ `MODEL_XFORM` - Shading is not supported for this driver. However, opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
  - □ `0` - Open the device without clearing the screen. This will set the color map mode to `CMAP_NORMAL`, but will not initialize the color map itself. In an X window, the color map that was associated with the window before `gopen` will be used by Starbase, without initialization.
  - □ `INIT` - Open and initialize the device as follows:
    1. Clear frame buffer to zeros.
    2. Reset the color map to its default values.
    3. Enable the display for reading and writing.
    4. In an X window, a new color map is created and initialized.
    5. hp98705 only- Download (if necessary) and initialize microcode for the transform engine.
  - □ `RESET_DEVICE` - Open and reset the device as follows:
    1. Clear image and overlay planes to zeros.
    2. Reset the image and overlay color maps to their default values.
    3. If opening the image planes, clear the overlay planes to the transparent color.
    4. hp98705 only - Download and initialize microcode for the transform engine.
    5. Enable the display for reading and writing.

Note The RESET_DEVICE flag bit should be used with caution, as it may adversely affect any other processes using the device. This flag bit is intended to reset a device completely and should only be necessary for devices in an unknown state, such as a device powered up in an external I/O space. Most programs should not use this flag bit. Programs opening X windows should never use this flag bit.

Syntax Examples

To open and initialize a PersonalVRX for output using the `hp98704` or `hp98705` drivers:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV,NULL,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,'',INIT);
```

Special Device Characteristics

For *device coordinate* (`dc`) operations, location (0,0) is the upper-left corner of the screen with x-axis values increasing to the right and y-axis values increasing down. The lower-right corner of the visible display is therefore (1279,1023).

Offscreen Memory Usage

Offscreen memory is managed by a global resource manager to insure that multiple processes do not encounter conflict. Offscreen memory is used by the device driver for:

- polygon fill patterns
- raster echo definitions (if software cursors are used)
- optimized raster fonts

- X Window system (if active, uses offscreen memory extensively)

Refer to the `gescape` `R_OFFSCREEN_ALLOC` for information on using the offscreen areas for application use.

Fast Alpha and Font Manager Functionality

The `hp98704` device driver supports raster text calls from the fast alpha and font manager libraries. These calls may be made while running in the overlay or image planes. Because raster fonts consist of one byte per pixel, image plane raster text is written only to the currently selected bank. This is similar to the operation of other raster functions, such as `block_write`. Fast alpha and font manager fonts can be optimized. See the **Fast Alpha/Font Manager Programmer's Manual** for further information.

The `hp98705` device driver does *not* support raster test calls from the fast alpha and font manager library.

Starbase Functionality

Commands Not Supported

The following commands are ignored on the `hp98704` device driver:

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

The following commands are not supported on the `hp98705` device driver:

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

Number of Light Sources

The `hp98705` device driver supports up to 15 light sources.

Commands Conditionally Supported

The following commands are supported on the `hp98704` device driver under the listed conditions:

`block_read,block_write`	The raw parameter for the `block_read` and `block_write` procedures is normally ignored by the `hp98704` device driver. To use the raw mode, you must call the `R_BIT_MODE gescape` discussed in this manual.
`pattern_define`	4x4 is the largest supported pattern.
`shade_mode`	The color map mode may be selected but shading can not be turned on `text_precision`. Only STROKE_TEXT precision is supported.
`vertex_format`	The use parameter must be zero, any extra coordinates supplied will be ignored.

`with_data`	`partial_polygon_with_data3d`
	`polygon_with_data3d`
	`polyhedron_with_data`
	`polyline_with_data3d`
	`polymarker_with_data3d`
	`quadrilateral_mesh_with_data`
	`triangle_strip_with-data`

7

Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

The following commands are supported on the `hp98705` device driver under the listed conditions:

`block_read,block_write`	The raw parameter for the `block_read` and `block_write` procedures is normally ignored by the `hp98705` device driver. To use the raw mode, you must call the `R_BIT_MODE gescape` discussed in this manual.

`inquire_fb_configuration`	An HP 98705 device running the `hp98705` device driver will report four `image_banks`, if 16 planes and dedicated Z-buffer are configured.

Note When the Z-buffer is installed in this way, Z-buffer access is possible through `block_write`, `block_read`, and `block_move`. The Z-buffer may be selected for read/write using `bank_switch`. The Z-buffer may not be displayed. The graphics accelerator cannot render to the Z-buffer.

`interior_style` If the polygon fill type is `INT_HATCH` then the following functionality will not work correctly:

- Hidden surface removal.
- Shading and lighting.
- Depth cueing.
- Back-facing attributes and culling.
- Splines, quadratic meshes, and triangle strips will not be hatched.

Note Performance is degraded in this mode.

`text_precision`	Only `STROKE_TEXT` precision is supported.

Gescapes

The hp98704 and hp98705 device drivers support the following gescape operations. Refer to Appendix A of this manual for details on gescapes.

- BLINK_INDEX—Blink an individual colormap index
- BLINK_PLANES—Blink the display (Blink rate is 375 Hz for this device)
- GR2D_PLANE_MASK—Enable multi-plane bit-per-pixel block reads and writes
- R_BIT_MASK—Bit mask
- R_BIT_MODE—Bit mode
- R_DEF_ECHO_TRANS—Define Raster Echo Transparency
- R_DEF_FILL_PAT—Define fill pattern
- R_ECHO_CONTROL—Control hardware cursor allocation
- R_ECHO_FG_BG_COLORS—Define cursor color attributes
- R_ECHO_MASK—Define a raster echo mask pattern
- R_FULL_FRAME_BUFFER—Full frame buffer
- R_GET_FRAME_BUFFER—Read frame buffer address
- R_LINE_TYPE—Define Line Style and Repeat Length
- R_LOCK_DEVICE—Lock device
- R_OFFSCREEN_ALLOC—Allocates offscreen frame buffer memory
- R_OFFSCREEN_FREE—Frees allocated offscreen frame buffer memory
- R_OV_ECHO_COLORS—Select Overlay Echo Colors
- R_OVERLAY_ECHO—Select Plane to Contain Cursor
- R_TRANSPARENCY_INDEX—Specify Transparency Index
- R_UNLOCK_DEVICE—Unlock device
- READ_COLOR_MAP—Read color map
- SET_REPLACEMENT_RULE—Set per-plane replacement rules
- SWITCH_SEMAPHORE—Semaphore control

The hp98705 device driver also supports the following gescape operations:

- GAMMA_CORRECTION - Enable/disable gamma correction
- LS_OVERFLOW_CONTROL - Sets options for lighting overflow situations
- PATTERN_FILL - Fill polygon with stored pattern
- POLYGON_TRANSPARENCY - Define front facing and backfacing transparency
- TRANSPARENCY - Allows *screen door* for transparency pattern

Performance Tips

`hp98704` and `hp98705`

1. If only one process is accessing the graphics display, then it is safe to turn off the semaphore operations (see the `SWITCH_SEMAPHORE gescape`). This may result in a 10 to 20 percent speedup. If a tracking process is initiated, then semaphores will automatically be turned on. Semaphores should never be off in a window environment, as there is always at least one additional process (the window server) trying to access the device.
2. The driver is able perform buffering to enhance performance. Performance may be degraded if `buffer_mode` is turned off, or if there are an inordinate number of calls to `make_picture_current`.
3. Performance optimizations have been made so that sequential calls of the same output primitive with no intervening attribute changes or different primitive calls perform faster. For example, the sequence of calls- `line_color polyline polyline` is optimized to perform faster than- `line_color polyline line_color polyline`. Grouping by primitive and subgrouping primitives by attribute can provide some performance improvements.

`hp98704`

1. If Starbase echos are overlayed (ie. in the fourth overlay plane), or hardware cursors are used, then graphics performance is significantly better since it is not necessary to "pick up" the cursor each time the frame buffer is updated.
2. Polygons are filled faster when the drawing mode is `SOURCE`, `NOT_SOURCE`, `ZERO`, or `ONE`.
3. Horizontal and vertical lines are often faster than diagonal lines on this device since the hardware block mover is used to generate pixels. The procedure `block_move` is faster than `block_read` or `block_write` since the hardware block mover can be used.
4. Performance of `block_read` and `block_write` is significantly better if both the source and destination begin on the same byte boundary, since data can be transferred 32 bits at a time rather than one byte at a time. For example, one way to ensure this condition is to define pixel arrays as type short (16-

bit integer) and then start `block_read` and `block_write` on even pixels only. This method can more than double performance

5. When the `R_BIT_MODE gescape` is called and the raw parameter is `TRUE`, each byte in a block read or write is interpreted as information for eight pixels, rather than just one. On some other HP displays, this mode runs no faster than sending a byte of information for each pixel. On the HP 98705 device, however, this bit-per-pixel mode runs significantly faster than byte-per-pixel block write. In this mode, only one bit of information is available for each pixel. By default, only one plane is written to during a bit-per-pixel transfer, specified by the `gescape R_BIT_MASK`. However, all opened planes can be written, each with their own replacement rule, if the `gescape SET_REPLACEMENT_RULE` has been properly called.

`hp98705`

1. Typically, the HP 98705 rendering engine renders primitives from its internal buffer in parallel with host CPU activity. Substantial performance enhancement can be realized by utilizing the parallel properties of the system. However, certain operations will cause the CPU to wait for the HP 98705 to finish emptying its buffer, for example, the `make_picture_current` operation. I/O intensive operations where the host is reading information from the HP 98705 may also cause this wait to occur.
2. For programs which use Z-buffer hidden surface removal with the dedicated Z-buffer, it is much faster to clear the Z-buffer simultaneously with screen clears than to do the clears sequentially. This is accomplished by calling `clear_control` with `CLEAR_ZBUFFER` ORed into the `mode` word. When this is done, subsequent calls to `clear_view_surface` and `dbuffer_switch` will clear the Z-buffer

Rendering (`hp98705`)

- When doing shaded polygons, the fewer the features, the faster the polygon generation. Positional viewpoint and light sources can degrade performance.
- With shading and Z-buffering off, the HP 98705 rendering engine runs at full speed, when rendering flat shaded polygons. Performance is noticeably slower when either or both rendering functions are enabled, especially when rendering large polygons.

- Using the pattern `gescape` or replacement rules that require extra reads of the frame buffer, for example, `source OR destination`, may also degrade performance.
- Rendering mode commands such as `hidden_surface`, `shade_mode`, and `double_buffer` can be slow. These should not be unnecessarily called. For example, it is not necessary to repeatedly call `hidden_surface` from an animation loop. These routines should be called to initialize a rendering mode and subsequently called again only to change the mode.

Raster Operations (hp98705)

- The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.
- The performance of `block_read` and `block_write` is significantly better if both the source and destination begin on the same byte boundary, since data can be transferred 32 bits at a time rather than one byte at a time. For example, one way to ensure this condition is to define pixel arrays as type short (16-bit integers) and then start block_read and block_write actions on even pixels only. This can more than double performance.
- When the `R_BIT_MODE gescape` is called and the raw parameter is `TRUE`, each byte in a block read or write is interpreted as information for eight pixels, rather than just one. On some other HP displays, this mode runs no faster than sending a byte of information for each pixel. On the HP 98705 device, however, this bit-per-pixel mode runs significantly faster than byte-per-pixel block write. In this mode, only one bit of information is available for each pixel. By default, only one plane is written to during a bit-per-pixel transfer, specified by the `gescape R_BIT_MASK`. However, all opened planes can be written, each with their own replacement rule, if the `gescape SET_REPLACEMENT_RULE` has been properly called.

Cautions

The following cautions are provided regarding use of the `hp98704` or `hp98705` driver:

1. As mentioned previously, accessing the off-screen portion of the frame buffer (using `gescape` calls) should be done with care, since other processes may use this region. See the `gescape` appendix entries on offscreen usage for details. The overlay off-screen contains the ITE font (which is regenerated when control-shift-reset is done on the ITE keyboard) and may contain any number of window systems fonts depending on the current window usage.
2. For the hp98705, polygons of up to 255 vertices (after clipping) are supported. If a polygon has more than 255 vertices, only the first 255 vertices are displayed.
3. Certain `gescapes` should be used with caution since they bypass protection mechanisms used to prevent multiple processes from conflicting with each other. For example, since the hardware resources can only be physically utilized by one graphics process at a time, the driver activates a semaphore and locks the device before doing any output. This ensures, for example, that process A will not change the replacement rule while process B is in the middle of filling a polygon. It also prevents the console (ITE) driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when done processing output. Some of the `gescapes` listed in this chapter allow the user to change this locking mechanism and should be used with great caution. In a windows environment, semaphores should not be turned off, and use of the locking mechanism can cause client timeouts or a system hang.

`hp98705`

1. Vertex color or intensity values should range between 0 to 32,767 when used in calls using DC values (e.g. `dcpolygon`).
2. If you are attempting to access the hardware directly while other processes are also using it (such as Starbase programs or window systems) then you must obey semaphore protocols and save/restore any hardware registers you alter. See the description of the `LOCK_DEVICE gescape` for details on semaphore

protocol. Use caution when manipulating protection mechanisms. Use of the locking mechanism in windows can cause client time outs or a system hang.

3. When using the HP 98705 device driver with a graphics accelerator it is possible for illegal operations to cause the transform engine or scan converter hardware to enter an unknown state. If this happens, Starbase will report an error the next time it tries to use the hardware. The user will see this as a transform engine timed out or hardware/scan_converter time out error. These are Starbase errors 14 and 52 respectively. If the HP 98705 device driver is being used, then this is a fatal error. When this error is discovered, Starbase reports the error and aborts execution.

The SRX Device

Device Description

The graphics display station includes a high resolution 19 inch color display, an HP 98720A Display Controller, and an optional graphics accelerator. The HP 98721A is an optional graphics accelerator for the HP 98720A controller. The display controller plugs into an I/O slot of the SPUs. See the "Introduction" section of this manual for systems supporting this controller.

Two device drivers are provided to access the HP 98720 display:

- HP 98720—used to access graphics windows with the X Window system or the graphics display without using the optional graphics accelerator.
- HP 98721—used to access the graphics display using only the optional graphics accelerator.

This section covers the HP 98720 and HP 98721 Device Drivers.

The display has a resolution of 1280 by 1024 pixels. The standard color display system has four planes of frame buffer to provide 16 simultaneous colors. You can add optional memory in banks of eight planes each. A fully configured system consists of three banks of frame buffer for 24 bit-per-pixel color, one bank for full Z-buffer capability (with graphics accelerator), and three overlay planes for non-destructive alpha, cursors, or graphics.

In order to use the HP 98721 Device Driver, the system must be configured with a graphics accelerator and at least one bank of eight planes. Four-plane systems are not supported with the graphics accelerator.

An 8-plane configuration allows 256 colors to be displayed simultaneously from a palette of 16 million. A 16-plane system is like two 8-plane frame buffers where only one 8-plane buffer is displayed at any time. This configuration is useful for double buffering. When three banks of frame buffer are installed, the system may be configured to display eight bits red, eight bits green, and eight bits blue per pixel. Double buffering may also be achieved at a resolution of four bits red, four bits green, and four bits blue.

The display system is a bit-mapped device with special hardware for:

- Write enable/disable individual planes.
- Video enable/disable individual planes.
- Memory writes with specified replacement rule. (see `drawing_mode` in the *Starbase Reference* manual)
- Video blinking of individual planes.
- Video blinking of individual color map locations.
- Arbitrary sized rectangular memory to memory copies.

The HP 98721 also provides:

- Write enable/disable of pixels in 4×4 cell for "screen door" transparency.
- Bit-slice processor with hardware floating point for high speed three-dimensional transformations.
- NMOS III scan converter with six axis interpolation for Gouraud shaded, Z-buffered vectors and polygons.

The display is organized as an array of bytes, with each byte representing a pixel on the display. With four planes installed, the four least significant bits of each byte determine the color, providing color map indices from 0 to 15. When eight planes are installed, color map indices range from 0 to 255. The color map is a RAM table that has 16 or 256 addressable locations and is 24 bits wide (8 bits each for red, green, and blue). Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

If you add optional banks of frame buffer memory to the minimal system, the four standard image planes function as overlay planes. These overlay planes have their own unique color map, separate from the color map used for the newly installed image planes. This color map consists of sixteen 4-bit entries. These four bits correspond to transparent, red, green, and blue (in order of Most Significant Bit (MSB) to Least Significant Bit (LSB). If the transparent bit (the MSB) is set to zero, the pixel color will be the color of the image planes "behind" the overlay planes. If the transparent bit is set to one, the pixel color is forced to the color specified by the red, green, and blue bits in the color map entry. Thus pixels in the overlay planes can be any combination of the primary colors or transparent.

You can use overlay planes for non-destructive alpha, graphics, or cursors. For example, when the HP 98720 is used on the system console, the Internal Terminal

Emulator (ITE) uses three of the overlay planes for alpha information. This way there is no interaction between ITE text and images in the graphics planes. The X Window system runs in configurations involving both image and overlay planes. See the *Starbase Graphics Techniques* for more information. To do graphics in the overlay planes, the HP 98720 or HP 98721 Device Driver may be opened directly to the overlay planes as if they were a separate device. Refer to the segment "Setting up the Device" in this section for more information.

One overlay plane is reserved for graphic cursors. When Starbase cursors are in the overlay plane performance is enhanced, since it it not necessary to "pick up" the cursor each time the frame buffer is updated. You can think of the overlay plane used for cursors as a separate cursor plane. Any data in the cursor plane will be displayed over data in the graphics planes. Data in the other three overlay planes will be displayed over data in the graphics planes and the cursor plane. For example, suppose a graphics application is running in the graphics planes while the window manager is running. If the application has a Starbase cursor in the overlay cursor plane, the cursor will always be visible inside regions of see-thru because the cursor has display priority over the graphics. If the cursor is moved outside the graphics window boundary, it is not visible since the window desktop environment is drawn to the overlay planes, which have display priority over the cursor plane.

Typically, the user does not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase does provide the `gescape` function `R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer. This `gescape` is discussed in the appendix of this manual. Frame buffer locations are then addressed relative to the returned address. The first byte of the frame buffer (byte 0) represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1279 is the last (right-most) pixel on the top line. The next 768 bytes of the frame buffer 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.

If more than one bank of optional frame buffer is installed, bank switching must be used to access the additional memory. A number of Starbase calls may set the bank register so it is advisable to call `bank_switch` just prior to making accesses to the frame buffer pointer to ensure desired results.

The off-screen portion of the frame buffer may be accessed via the `gescape` function `R_FULL_FRAME_BUFFER`. Use this `gescape` carefully since other processes and Starbase access the frame buffer off-screen memory.

Series 800 On the Series 800 computers, a write to I/O space must be on the word boundaries. The frame buffer is mapped as an integer (32 bits) per pixel. Therefore, when you write directly to the frame buffer on the HP 98720 Graphics Display Station, each pixel is written with an integer access.

If writing to the HP 98720 image buffer and not in `CMAP_FULL` color map mode, only one bank can be written at a time. The bank to be written must be established by a call to `bank_switch`. Then, to write to bank *n*, place the pixel value to be written into byte *n* of the interger, where *n* can be 0, 1, or 2, and the LSB is byte 0.

All three banks for one pixel can be written simultaneously by packing all three bank values for the pixel into the integer value and having the color map mode as `CMAP_FULL` before writing.

Setting Up the Device On Series 300

The HP 98720 or HP 98721 Device Drivers can be used if the display is configured in either internal or external address space. Refer to the *Configuration Reference Manual* for a description of internal and external address space.

Note If the HP 98720 is configured as an external display, there will not be an Internal Terminal Emulator (ITE) for that device. Since it is the ITE that normally initializes the display, external devices must be reset after power-up by running a simple Starbase program with a mode of `RESET_DEVICE` in the `gopen` call. It may also be necessary to run this program after running an application which manipulated the overlay color map, such as windows. An example program, which could be called from `/etc/rc` during power-up, is given at the end of this section. For more details concerning the effects of `RESET_DEVICE`, see the "Device Initialization" segment of this section.

For the HP 98721, the Graphics Interface card may be installed in any DIO slot in the computer's backplane or in any I/O slot of the expander.

Switch Settings

The Graphics Interface card has a single 6-bit address select switch. One bit, labeled `FB`, determines the frame buffer location, while the other five switch bits, labeled `CS`, determine the location within the DIO memory map of the HP 98720 control space. Silkscreening on the printed circuit board indicates the meaning of the bits.

The frame buffer consumes two Mbytes of I/O address space, starting at `FB_BASE`. The switch bit labeled `FB` determines the address of `FB_BASE` as shown below.

Table 8-1. HP 98720 Frame Buffer Locations

FB	FB_BASE (hex)
0	$200000
1	$800000

Typical systems will map the frame buffer to $200000. However, some systems which have multiple displays may map the frame buffer address to $800000. When the frame buffer address is set to $800000, the HP Series 300 Model 320 SPU memory limit is reduced from 7.50 Mbytes to 5.75 Mbytes. This occurs since the frame buffer is mapped into the upper two Mbytes of memory address space.

The control space requires 128 Kbytes starting at CTL_BASE. The five switch bits labelled CS, determine the address of CTL_BASE. The HP 98720 may be configured as an external or internal display. Since only 64 Kbytes are normally allotted for external I/O select codes, two consecutive select codes will be consumed if the device is configured as an external display. The control space may be located at any of 32 positions. Sixteen positions are reserved in internal I/O space and sixteen are in external I/O space (with five reserved). The table below lists the binary switch setting with the corresponding values of CTL_BASE for external I/O settings, as well as the select code spaces consumed.

Table 8-2. Control Space Settings (External I/O)

CS Setting	CTL_BASE (hex)	Select Codes
01011	$560000	
10101	$6A0000	10-11
10110	$6C0000	12-13
10111	$6E0000	14-15
11000	$700000	16-17
11001	$720000	18-19
11010	$740000	20-21
11011	$760000	22-23
11100	$780000	24-25
11101	$7A0000	26-27
11110	$7C0000	28-29
11111	$7E0000	30-31

If the HP 98720 is configured as the system console, CTL_BASE needs to be placed at $560000, which is an internal I/O setting. If the device is not used as the system console, the control space should not be placed in internal I/O space, since it is likely to overlap the address space of other system hardware. In this case, an external I/O space setting should be selected with two consecutive select codes which are unused by the system.

Example Program To Reset the SRX

The following example uses the HP 98720 device driver. You can substitute the HP 98721 device driver.

```
/*
 * Starbase program: reset98720.c
 * Compile: cc -o reset98720 reset98720.c -ldd98720 -lsb1 -lsb2 -lm
 * Destination: /usr/bin
 * Execute: add line to the /etc/rc - "/usr/bin/reset98720 /dev/crt.external"
 *
 * Example program to be put in /etc/rc for resetting an external HP 98720
 * device during power-up.
 */
 #include <starbase.c.h>

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

    if ((fildes = gopen(argv[1],OUTDEV,"hp98720",INIT|RESET_DEVICE)) < 0)
         printf("External HP 98720 %s initialization failed.\n",argv[1]);
    else
       printf("External HP 98720 %s initialization succeeded.\n",argv[1]);
       gclose(fildes);
    }
}
```

Setting Up the Device On Series 800

Up to four HP 98720 or four HP 98721 devices can be connected to a Series 800 Model 825 or 835 SPU. However, it is recommended that only two HP 98720 of HP 98721 devices have the Internal Terminal Emulator (ITE) or window systems running on them.

Only one HP 98720 device or HP 98721 can be connected to a Series 800 Model 840.

Special Device Files (mknod) On Series 300

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod`(1M) information in the *HP-UX Reference* for further details. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however the name that is suggested for the devices is `crt`.

The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the `mknod` command.

When the device is at the internal address (refer to the "Switch Settings" section) the `mknod` parameters should create a character special device with a major number of 12 and a minor number of 0. Note that the leading `0x` causes the number to interpreted hexidecimally.

```
mknod /dev/crt c 12 0x000000
```

When the device is at an external address (refer to the "Switch Settings" section) the `mknod` parameters should create a character special device with a major number of 12 and a minor number of 0x⟨*sc*⟩200 where *Sc* is the two-digit external select code. Note that the leading `0x` causes the number to be interpreted hexidecimally.

```
mknod /dev/crt c 12 0x⟨sc⟩0200
```

8

The HP 98720 Device Driver may also be opened to the overlay planes in graphics mode if they are present. Since one plane is still reserved for cursors, the graphics device will look like a three plane device in this mode. Since the terminal emulator and window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is opened in this manner while these processes are present. To open the HP 98720 or HP 98721 Device Driver to the overlay planes instead of the graphics planes, the last byte of the minor number must be 1.

For example, when the device is at an internal address, the `mknod` parameters for the overlay device should create a character special device with a major number of 12 and a minor number of 1.

```
mknod /dev/ocrt c 12 0x00001
```

When the device is at an external address (refer to the section on "Switch Settings") the `mknod` parameters for the overlay device should create a character special device with a major number of 12 and a minor number of 0x⟨*sc*⟩0201 where ⟨*sc*⟩ is the two-digit external select code.

```
mknod /dev/ocrt c 12 0x⟨sc⟩0201
```

Special Device Files (mknod) On Series 800

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod`(1M) information the *HP-UX Reference* for further details. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however the names that are suggested for the devices are `crt`, `crt0`, `crt1`, or `crt2`.

The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the `mknod` command.

When creating the device file the `mknod` parameters should create a character special device with a major number of 14 and a minor number of the format below (where ⟨*lu*⟩ is the two digit hardware logical unit number). Note that the leading `0x` causes the number to be interpreted hexadecimally.

```
mknod /dev/crtx c 14 0x00⟨lu⟩00
```

The HP 98720 or HP 98721 Device Driver may also be opened to the overlay planes in graphics mode if they are present. Since one plane is still reserved for cursors, the graphics device will look like a three plane device in this mode. Since the terminal emulator and window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is opened in this manner while these processes are present. To open the HP 98720 or HP 98721 Device Driver to the overlay planes instead of the graphics planes, the last byte of the minor number must be 1.

For example, the `mknod` parameters for the overlay device should create a character special device with a major number of 14 and a minor number of the format indicated below (where ⟨*lu*⟩ is the two digit hardware logical unit number):

```
mknod /dev/ocrtx c 14 0x00⟨lu⟩01
```

Linking the Driver

Shared Libraries

The shared HP 98720 Device Driver is the file named `libdd98720.sl` in the `/usr/lib` directory. The shared HP 98721 Device Driver is the file named `libdd98721.sl` in the `/usr/lib` directory. The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
   -lXhp11 -lX11 -ldld -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Upon device initialization the proper driver will be loaded. See the discussion of the `gopen` procedure in the **Device Initialization** section of this chapter for details.

Archive Libraries

The archive HP 98720 Device Driver is the file `libdd98720.a in the /usr/lib` directory. The archive HP 98721 Device Driver is the file `libdd98721.a in the /usr/lib` directory. The device driver may be linked to a program using the absolute path name, for example, `/usr/lib/libdd98720.a`, an appropriate relative path name, or by using the `-l` option as in `-ldd98720` with the `LDOPTS` environmental variable set to `-a archive`.

The reason for using the `LDOPTS` environmental variable is that the `-l` option will look for a shared library driver first and then look for the archive driver if shared was not found. By exporting the `LDOPTS` variable as specified above, the `-l` option will only look for archive drivers. For more information, refer to the *Programming on HP-UX* manual on linking shared or archive libraries.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with the HP 98720 driver, use:

```
   export LDOPTS="-a archive"
```

and then:

```
   cc example.c -ldd98720 -L/usr/lib/X11R5 -lXwindow\
   -lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
   F77 example.f -ldd98720 -Wl,-L/usr/lib/X11R5 -lXwindow\
    -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
   pc example.p -ldd98720 -Wl,-L/usr/lib/X11R5 -lXwindow\
    -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Device Initialization

Parameters for gopen

Note Because the transform engine is not multi-tasking, only *one* HP 98721 driver may be opened to a device file. Other HP 98720 drivers may be opened to that device file if multiple Starbase drivers are needed.

The `gopen` procedure has four parameters: Path, Kind, Driver, and Mode.

Path The name of the special device file created by the `mknod` command as specified in the last section (for example, `/dev/crt`.)

Kind Indicates the I/O characteristics of the device. This parameter must be `OUTDEV` for this driver unless a window is being opened, in which case it may be `OUTINDEV`.

Driver The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire the device and by default load the accelerated driver (if applicable). For example:

`NULL`	for C
`char(0)`	for FORTRAN77
''	for Pascal

Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is ignored. For example:

`"hp98720"`	*for C.*
`'hp98720'//char(0)`	*for FORTRAN77.*
`'hp98720'`	*for Pascal.*

Mode	The mode control word consisting of several flag bits which are *or* ed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure. See the *Starbase Graphics Techniques* for a description of the `gopen` function of an X window.

- `SPOOLED`—cannot spool raster devices.
- `MODEL_XFORM`—HP 98720 only—Shading is not supported for this device. However, opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
- `0`—open the device, but do nothing else. The software color map is initialized on monochrome monitors.
- `INIT`—open and initialize the device as follows:

1. Clear frame buffer to 0s.
2. Reset the color map to its default values.
3. Enable the display for reading and writing.
4. HP 98721—Download the fransform engine's microcode.

- `RESET_DEVICE`—open and reset the device as follows:
  1. Clear frame buffer and overlays to 0s.
  2. Reset the color map to its default values.
  3. Clear the overlay color map.
  4. Enable the display for reading and writing.
  5. Reset the graphics accelerator.

  Note that the `RESET_DEVICE` flag bit should be used with caution: it will adversely affect any other processes using the device. This flag bit is intended to reset a device completely. This should only be necessary for devices in an unknown state such as a device powered up in an external I/O space. Most programs should not use this flag bit.

Syntax Examples

To open and initialize an HP 98720 or HP 98721 device for output:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0), OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,,INIT);
```

Special Device Characteristics

For Device Coordinate operations, location (0, 0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is therefore (1279, 1023).

Offscreen Memory Usage

Each time the HP 98720 Device Driver is opened it allocates a portion of offscreen memory. This is used for fill pattern storage, raster echo definitions, and other functions. The size of the areas used are 64×192 pixels and 32×4 pixels. If the driver has been opened to the overlay planes, the offscreen area used is in the overlay planes; otherwise the area used is in the image planes. Up to ten of these offscreen areas may be allocated. One is reserved for the HP 98721 Device Driver and the other nine are for HP 98720 Device Drivers. This means that no more than one HP 98721 Device Driver and nine HP 98720 Device Drivers may be opened to a device at the same time. If nine HP 98720 Device Drivers are opened to the image planes, another nine may be opened to the overlay planes. However, only one HP 98721 Device Driver may be opened to a device at any time to either image or overlay planes. The X11 server uses a similar cursor area and also uses offscreen for client pixmaps. Accessing offscreen memory while the X11 server is running is not recommended.

See the HP 98721 Device Driver section for more information on offscreen memory usage.

Device Defaults

Number of Color Planes

When the `gopen` procedure is called, this driver asks the device for the number of color planes available. This number can be either 3 (for the overlay planes) or 4, 8, 16, or 24 (for the image planes). The device driver then acts accordingly.

Dither Default

The default number of colors searched for in a dither cell is 2. The number of colors allowed in a dither cell is 1, 2, 4, 8 or 16. For devices having 24 or more planes in `CMAP_FULL` mode (see `shade_mode`) dithering is not supported since full 24-bit color is available. If you are double buffering with 12 planes per buffer then the number of colors allowed in a dither cell is 1, 2, or 4.

Raster Echo Default

The default raster echo is the 8×8 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 allowed for a raster echo is 64×64 pixels. The default drawing mode for the raster echo is 7 (a logical *OR*).

By default the raster echo is written to the graphics planes. All other echo types are written to an overlay plane. The location of raster and non-raster echoes can be changed using the `gescape` function `R_OVERLAY_ECHO`.

Color Planes Defaults

The default configuration is a 4- or 8-plane color mapped system regardless of the number of frame buffer banks installed.

All planes in the first bank are display enabled and write enabled.

Semaphore Default

Semaphore operations are enabled.

Line Type Defaults

The default line types are created with the bit patterns shown below:

Table 8-3.

Line Type	Pattern
0	1111111111111111
1	1111111100000000
2	1010101010101010
3	1111111111111010
4	1111111111101010
5	1111111111100000
6	1111111111110110
7	1111111110110110

Default Color Map

If the fourth `gopen` parameter is zero (`0`), the current hardware color map is used on color displays.

If the fourth gopen parameter is INIT, the current color map is initialized to the default values shown below.

Table 8-4. 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

Use the inquire_color_map procedure to see the rest of the 255 colors.

When INIT is used in the shade_mode procedure call the color map initialization is based on the value of the mode parameter and the number of frame buffer banks installed.

CMAP_NORMAL	Only one bank of the first two can be displayed at a time. If a third bank is installed it can not be displayed in this mode.

`CMAP_MONOTONIC` The color map will be initialized as:

```
for (i=0; i<256; i++) {
  cmap[i].red = cmap[i].green = cmap[i].blue = i/255.0;
}
```

Only one bank of the first two can be displayed at a time. If a third bank is installed it can not be displayed in this mode.

`CMAP_FULL` With less than three banks installed the color map will be initialized as three bits red, three bits green and two bits blue. The three most significant bits are red and the two least significant bits are blue. Only one bank of the first two can be displayed at a time.

With three or more banks installed the color map will be initialized as the `CMAP_MONOTONIC` case above but now the first bank of eight will go through the blue portion of the color map, the second bank goes through the green portion, and the third bank goes through the red portion. In this mode the color map is transparent and the eight bits from each bank drives the appropriate video color level. The color map could be subsequently modified in this mode to perform functions like gamma correction or double buffering of four bits per color.

Red, Green, and Blue

Each file descriptor opened as an output device has a color table associated with it. If multiple file descriptors are open to the same device, the color table and the device's color map may not always be identical. The color table does not track the color map if the device's color map is changed by another file descriptor path.

For Starbase procedures that have parameters for red, green, and blue, the way the actual color is chosen depends on the current `shade_mode` setting.

`CMAP_NORMAL` The color map is searched for the color that is closest in RGB space to the one requested. That color map index is written to the frame buffer for subsequent output primitives. It is more efficient to select a color with an index rather than specifying a color with red, blue, and green values in this mode because

	it takes extra time to figure out which index in the color table most closely matches the specified color.
`CMAP_MONOTONIC`	The red, green, and blue value is converted to an intensity value using the equation: `0.30*red+0.59*green+0.11*blue` This intensity is converted to an index value by mapping intensity 0.0 to the minimum index set by `shade_range`, and intensity 1.0 to the maximum index set by `shade_range`. This mode is useful for displaying a high-quality monochrome picture on an 8-plane system from data that produces a high quality color picture on a 24-plane system.
`CMAP_FULL`	The color values will be mapped directly to an index with the assumption the color map is initialized to a predefined full color state.

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 8-5. SRX Display Types

SRX [FB0][1]	HP 98720	High-Res Color
SRX [FB1]	HP 98720	High-Res Color
SRX [FB2]	HP 98720	High-Res Color
SRX [FB3]	HP 98720	High-Res Color

[1] This system comes standard with four planes, with optional banks of eight planes each. The four standard planes become the overlay planes if any additional banks are loaded.

The supported server modes are Overlay and Stacked Screen.

Table 8-6. Windows in Overlay Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
3	No	PseudoColor	●	●	one color reserved for transparency in combined mode

The supported server modes are Image and Stacked Screen.

Table 8-7. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	
16	No				Not supported
	Yes (8/8)	PseudoColor	●		only on FB2, FB3
24	No	DirectColor	●		only on FB3
	Yes (12/12)	DirectColor	●		only on FB3

X11 Cursors and Starbase Echos

The following list shows default positions where the Starbase echo and X11 cursor (called echo and cursor, respectively) reside for each of the X11 server operating modes.

HP 98720 Display

- Overlay Mode

  If X11 window overlay-plane is opened, echo shares three overlay planes.

  If image-planes are opened, raster echo resides in image planes.

  If image-planes are opened in raw mode, vector echo resides in cursor plane.

  Cursor shares three overlay planes.

- Image Mode

  If X11 window overlay-plane is opened:

  - raster echo resides in image planes.
  - vector echo resides in cursor planes.

  Cursor shares image planes.

- Stacked Screen Mode

  If overlay-plane in X11 windows is opened, echo shares three overlay planes.

  If image-plane in X11 windows is opened:

  - raster echo resides in image planes.
  - vector cursor resides in cursor plane.

  Cursor:

  - shares image plane for image-plane window.
  - shares overlay plane for overlay-plane window.

Starbase Functionality

Commands Not Supported

The following commands are not supported on the HP 98720 nor HP 98721. If one of these commands is used by mistake, it will be ignored and not cause an error.

```
alpha_transparency               light_attenuation
bf_alpha_transparency            line_filter
bf_control                       perimeter_filter
bf_fill_color                    set_capping_planes
bf_interior_style                set_model_clip_indicator
bf_perimeter_color               set_model_clip_volume
bf_perimeter_repeat_length       surface_coefficients
bf_perimeter_type                texture_index
bf_surface_model                 texture_viewport
bf_texture_index                 texture_window
contour_enable
define_contour_table
define_texture
deformation_mode
depth_cue_color
depth_cue_range
hidden_surface
interior_style (INT_OUTLINE)
interior_style (INT_POINT)
```

In addition, the HP 98720 does not support the following commands:

```
backface_control
bf_coefficients
define_trimming_curve
depth_cue
hidden_surface
light_ambient
light_attenuation
light_model
```

```
light_source
light_switch
shade_range
surface_model
viewpoint
zbuffer_switch
```

Commands Conditionally Supported

The following commands are supported on the HP 98720 and HP 98721 under the listed conditions:

`bank_switch`	Only Bank 0 and Bank 1 can be selected for the ⟨*dbank*⟩ parameter.
`block_read, block_write`	The "`raw`" parameter for the `block_read` and `block_write` commands is normally ignored by this device driver. To use the raw mode, you must call the `gescape` function `R_BIT_MODE` discussed in the appendix of this manual.
`pattern_define`	4×4 is the largest supported pattern.
`shade_mode`	The color map mode may be selected but shading can not be turned on.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	The `use` parameter must be zero, any extra coordinates supplied will be ignored.

The following commands are supported on the HP 98721 under the listed conditions:

`backface_control`	Backface color is supported only if shading is on as set by `shade_mode`. Also, `backface_color` does not work correctly for spline surfaces when `VERTEX_FORMAT` specifies normal per vertex.
`interior_style`	If the polygon fill is `INT_HATCH`, the following functionality will not work correctly.

- Hidden surface removal.
- Shading and lighting.
- Depth cueing.
- Backfacing attributes and culling.
- Splines, quadralateral meshes, and triangle strips, will not be hatched.

  Performance is also degraded in this mode.

`light_model` `light_source` `light_switch`	Supports nine light sources; one ambient and eight positional or directional.
`with_data`	`partial_polygon_with_data3d` `polygon_with_data3d` `polyhedron_with_data` `polyline_with_data3d` `polymarker_with_data3d` `quadrilateral_mesh_with_data` `triangle_strip_with-data` Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

Splines

`QUARTIC` and `QUINTIC` splines (i.e., fifth- and sixth-order splines) are not supported on the HP 98721 driver.

The commands that are affected are `spline_curve2d`, `spline_curve3d`, and `spline_surface`.

Fast Alpha and Font Manager Functionality

The HP 98720 Device Driver supports raster text calls from the fast alpha and font manager libraries. These calls may be made while running in the overlay or image planes. See the *Fast Alpha/Font Manager Programmer's Manual* for further information.

The HP 98721 Device Driver does *not* support raster text calls from the fast alpha and font manager library.

Parameters for gescape

The HP 98720 and HP 98721 support the following gescape operations. Refer to Appendix A of this manual for details on gescapes.

- `BLINK_INDEX`—Alternate between HP 98720 hardware color maps.
- `BLINK_PLANES`—Blink display (blink rate is 3.75 Hz for this device).
- `R_BIT_MASK`—Bit mask.
- `R_BIT_MODE`—Bit mode.
- `R_DEF_ECHO_TRANS`—Define raster echo transparency.
- `R_DEF_FILL_PAT`—Define fill pattern.
- `R_FULL_FRAME_BUFFER`—Full frame buffer.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_LINE_TYPE`—Define line style and repeat length.
- `R_LOCK_DEVICE`—Lock device.
- `R_OV_ECHO_COLORS`—Select overlay echo colors.
- `R_OVERLAY_ECHO`—Select plane to contain cursor.
- `R_TRANSPARENCY_INDEX`—Specify HP 98720 transparency index.
- `R_UNLOCK_DEVICE`—Unlock device.
- `READ_COLOR_MAP`—Read color map.
- `SWITCH_SEMAPHORE`—Semaphore control.

The HP 98721 also supports the following gescape operations:

- `LS_OVERFLOW_CONTROL`—Sets options for overflow situations.
- `PATTERN_FILL`—Fill polygon with stored pattern.
- `TRANSPARENCY`—Allow "screen door" for transparency pattern.
- `ZBUFFER_ALLOC`—Allocates frame buffer memory for Starbase.

- ZSTATE_RESTOR—Allows creation of 3D cursors in overlay.
- ZSTATE_SAVE—Allows creation of 3D cursors in overlay.
- ZWRITE_ENABLE—Allows creation of 3D cursors in overlay.

Performance Tips

1. As with any driver, buffering is done to enhance performance. Performance can be degraded substantially if `buffer_mode` is turned off or an inordinate amount of `make_picture_current` calls are done.
2. Performance optimizations have been made so that sequential calls of the same output primitive with no intervening attribute changes or different primitive calls go faster. For example, the sequence `polygon, polygon, polyline, polyline` is faster then `polygon, polyline, polygon, polyline`. Also `line_color, polyline, polyline` is faster than `line_color, polyline, line_color, polyline`. So grouping by primitive and subgrouping primitives by attribute can give some performance improvements.
3. If Starbase echoes are in the overlay plane, graphics performance is significantly better since it is not necessary to "pick up" the cursor each time the frame buffer is updated.
4. Screen clears will be significantly faster if the area to be cleared starts on a 128-pixel boundary and is some multiple of 128-pixels wide. This can be checked by using the Starbase routines `transform_point` and `vdc_to_dc` to convert the bounds of the clear rectangle to device coordinates. Screen clears to the default `vdc_extent` will be aligned. Screen clears are also much faster when the background color index is zero. Screen clears with a non-zero index require two passes resulting in slower performance.
5. The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.
6. Performance of `block_read` and `block_write` is significantly better if both the source and destination begin on the same byte boundary (since data can be transferred 32 bits at a time rather than one byte at a time). For example, one way to ensure this condition is to define pixel arrays as type short (16-bit integers) and then start `block_read` and `block_write` on even pixels only. This can more than double performance.

hp98720

1. If only one process is accessing the graphics display, it is safe to turn off the semaphore operations. See the `SWITCH_SEMAPHORE gescape`. With semaphores turned off you can increase your program's speed 10 to 20 percent. If a tracking process is initiated, semaphores will automatically be turned on. See "Cautions" below for more information.
2. Polygons are filled faster when the drawing mode is ⟨*source*⟩, ⟨*not_source*⟩, `ZERO`, or `ONE`.
3. Horizontal and vertical lines are faster than diagonal lines on this device since the hardware block mover is used to generate the pixels.

hp98721

1. If only the HP 98721 driver and the ITE are accessing the graphics device it is safe to turn off the semaphore operations. This can result in a 10 to 20 percent speed increase. If a tracking process is initiated, semaphores will automatically be turned on. While in most cases the system will work with the tracking process running and semaphores turned off, there is a chance that continuous movement of the cursor could halt the graphics accelerator for a significant period of time. If this is not a problem, semaphores may be turned off after tracking is initiated.
2. When drawing shaded polygons, the fewer the features, the faster the polygon generation. Positional viewpoint and light sources can significantly degrade performance.
3. With dithering, shading, and Z-buffering off, the SRX rendering engine runs at full speed while rendering flat shaded polygons. These three rendering techniques slow the rendering of polygons on the SRX. This is especially noticeable on large polygons. Turning on any one of the three could noticeably lower the rendering performance.

   When using the full 24 planes, dithering is turned off by default, and 12/12 double buffering will turn dithering on by default. To turn dithering off again, use `fill_dither` (`fildes`, 1).

Using the `PATTERN_FILL gescape` or replacement rules that require extra reads of the frame buffer (e.g. ⟨*source*⟩ *or* ⟨*destination*⟩) will also degrade performance. It takes time to do the extra reads.

4. Typically, the SRX rendering engine renders primitives from its internal buffer as the system CPU is doing other things. Substantial performance benefits can be realized from this parallel processing.

   However, certain operations will cause the CPU to wait for the SRX to finish emptying its buffer. An example of this wait is the `make_picture_current` operation. Also, any operation that reads information from the SRX will cause this wait to occur. Following are some typical operations that read values from the SRX:

   a. Many two-dimensional primitives used in three-dimensional mode read the Z value from the SRX. The following primitives are examples: `text2d`, `polymarker2d`, `arc`, `ellipse`, and `spline_curve2d`. The solution is to always use three-dimensional primitives when in three-dimensional mode.

   b. Two operations read the matrix values from the SRX: `pop_matrix2d` and `pop_matrix3d`. If the values in the popped matrix are not needed, use `pop_matrix`, which does not cause any information to be read from the SRX.

Cautions

The following cautions are provided in using the `hp98720` and `hp98721` drivers:

1. As mentioned previously, accessing the off-screen portion of the frame buffer (using the `gescape` functions) should be done with care since other processes access this region. The HP 98720 and HP 98721 drivers use a 128×1024 strip of off-screen memory that begins at (1920, 0). The HP 98721 driver in particular uses the rectangular area of 64×196 located at (1984, 828). This area is used to store the fill pattern when in `CMAP_NORMAL` mode and three 64×64 areas for storing the raster cursor, raster cursor transparency pattern, and the saved raster. If the HP 98721 driver is not being used in `CMAP_NORMAL` mode, raster cursors are not being used in the image planes and no HP 98720 drivers are opened to the image planes, the area can be safely used for more Z-buffer or other purposes. If the HP 98721 driver is opened to the overlay planes, it is not recommended that any of the overlay off-screen be used. The overlay off-screen contains the ITE font (which is regenerated when control-shift-reset is done on the ITE keyboard) and may contain any number of window system fonts depending on the current window usage.
2. Certain `gescape` functions should be used with caution since they bypass protection mechanisms used to prevent multiple processes from interfering with each other. For example, since the hardware resources can only be rationally used by one graphics process at a time, the driver activates a semaphore and locks the device before doing any output. This ensures, for example, that process A will not change the replacement rule while process B is in the middle of filling a polygon. It also prevents the terminal (`tty`) driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when done processing output. Some of the `gescape` functions listed in the appendix for this manual allow the user to change this locking mechanism and should be used with *great caution*. Semaphores should never be turned off when operating in a window environment.
3. When using the HP 98720 device with a graphics accelerator it is possible for illegal operations to cause the transform engine or scan converter hardware to enter an unknown state. If this happens, Starbase will report an error the next time it tries to use the hardware. The user will see this as a `Transform engine timed out` or `Hardware/scan_converter time out` error. These are Starbase errors 14 and 52 respectively. This is a very serious error condition.

If the HP 98721 device driver is being used, then this is a fatal error. When this error is discovered, Starbase reports the error and aborts execution.

If an application needs to take some emergency action before an untimely termination, (such as saving valuable data) the application should check for these error conditions and take appropriate measures. Errors may be caught by an application using the `gerr_control` procedure described in the *Starbase Reference* manual.

It is also possible to avoid the termination completely if the application's error handler does not return control to Starbase. It is impossible, however, to proceed with any graphics efforts using the accelerator.

If the HP 98720 driver is being used to access the hardware and if it detects such an error; it will report the error condition, reset the transform engine, and continue (since it does not use the accelerator hardware).

4. If a single device is opened with each of these two devices, HP 98720 and HP 98721, the cursor may not be fully erased between updates.

9

The TurboSRX Device

Device Description

This graphics display station includes an HP 98730A Graphics Controller, a high resolution 16 or 19 inch color display (purchased separately), an optional accelerator and Z-buffer, and optionally 8, 16, or 24 planes of frame buffer memory. The HP 98731A is the optional graphics accelerator and Z-buffer for the HP 98730A Display Controller. The graphics controller plugs into an I/O slot of the SPUs. See the "Introduction" section of this manual for systems supporting this controller.

Two device drivers are provided to access the TurboSRX display:

- HP 98730—The HP 98730 Device Driver is used to access the graphics display without using the optional graphics accelerator. Access can be with or without the X Window System.
- HP 98731—The HP 98731 Device Driver is used to access the graphics display using only the optional graphics accelerator, with or without the X Window System.

This section covers the HP 98730 and HP 98731 Device Drivers.

The display has a resolution of 1280×1024 pixels. The standard color display system has eight planes of frame buffer to provide 256 simultaneous colors. You can add optional memory in banks of eight planes each. A fully configured system consists of three banks of frame buffer for full 24 bit per pixel color, dedicated boards for full 16 bit Z-buffer capability with graphics acceleration, and four overlay planes for non-destructive alpha, cursors, or graphics.

An 8-plane configuration allows 256 colors to be displayed simultaneously from a pallet of 16 million. A 16-plane system is like two 8-plane frame buffers where only one 8-plane buffer is displayed at any time. This configuration is useful for double buffering. When three banks of frame buffer are installed, the system may be configured to display eight bits red, eight bits green and eight bits blue per pixel. Double buffering may also be achieved at a resolution of four bits red, four bits green and four bits blue.

The display system is a bit-mapped device with special hardware for:

- Write enable/disable individual planes.
- Video enable/disable individual planes.
- Memory writes with specified replacement rule (see `drawing_mode` in *Starbase Reference* manual).
- Video blinking of individual planes.
- Video blinking of individual color map locations.
- Arbitrary sized rectangular memory to memory copies.
- Pixel pan and zoom.
- Analog blending of frame buffer outputs.
- Raster and vector cursors.

The HP 98731 also provides the following features:

- Write enable/disable of pixels in 4×4 cell for "screen door" transparency.
- Up to three VLSI NMOS III processors with hardware floating point for high speed three-dimensional transformations.
- NMOS III scan converter with six axis interpolation for Gouraud shaded, Z-buffered vectors and polygons.
- Pixel clipping for full speed graphics to obscured windows.
- Dedicated 2K by 1K 16-bit zbuffer.

The display is organized as an array of bytes, with each byte representing a pixel on the display. (On Series 800 systems the display can be accessed on a 32-bit word/pixel basis.) When eight planes are installed, color map indexes range from 0–255. The color map is a RAM table that has 16 or 256 addressable locations and is 24 bits wide (eight bits each for red, green and blue). Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

In addition to the frame buffer banks of eight planes each, four overlay planes are provided. These overlay planes have their own unique color map, separate from the color map used for the image planes. This color map consists of sixteen 24-bit entries, allowing you to select sixteen colors from the full pallette of over 16 million choices. In addition, each entry in the overlay color map may be set to be `dominant`, `non-dominant`, or `blended` with the image planes.

A `dominant` entry causes all pixels in the overlays set to that value to display the color in the overlay map, regardless of values in the image planes "below" it.

A `non-dominant` entry causes pixels with that value to display the color in the image planes "below".

A `blended` entry will cause the analog color output from the overlays to be summed with the analog output from the image planes. Color values are set to a maximum value of 1.0 if the sum would exceed this saturation value.

By default, the HP 98731 Device Driver sets all overlay color map entries to be `dominant` when opened to the overlays. Entries may be set to be `non-dominant` with the Starbase `gescape R_TRANSPARENCY_INDEX`. Entries may be set to blend with the image planes by using the Starbase `gescape OVERLAY_BLEND`. See the descriptions of these `gescape` functions for more details.

You can use overlay planes for non-destructive alpha, graphics, or cursors. For example, on displays that run it, the Internal Terminal Emulator (ITE) uses three of the overlay planes for alpha information. This way there is no interaction between ITE text and images in the graphics planes. The X Window system uses both the overlay and image planes. To draw in the overlay planes the HP 98730 Device Driver or HP 98731 Device Driver may be opened directly to the overlay planes as if they were a separate device (refer to "Setting up the Device" in this chapter for more information).

The TurboSRX provides one hardware cursor which supports all Starbase echo types. If more than one cursor is needed, one overlay plane can be used for graphic cursors.

Typically, you do not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase does provide the `gescape` function `R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer (this `gescape` is discussed in the appendix of this manual). Frame buffer locations are then addressed relative to the returned address. The first byte of the frame buffer (byte 0) represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1279 is the last (right-most) pixel on the top line. The next 768 bytes of the frame buffer 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.

If more than one bank of optional frame buffer is installed then bank switching must be used to access the additional memory. A number of Starbase calls may set the bank register so it is advisable to call `bank_switch` just prior to making accesses to the frame buffer pointer to ensure desired results.

If you are attempting to access the hardware directly while other processes are also using it (such as Starbase programs or window systems), you must obey semaphore protocols and save/restore any hardware registers you alter. See the description of the `LOCK_DEVICE` `gescape` for details on semaphore protocol.

The off-screen portion of the frame buffer may be accessed via the `gescape` function `R_FULL_FRAME_BUFFER` also documented in the appendix. Care should be taken when using this `gescape` since other processes, Starbase, and the window system access the frame buffer off-screen memory.

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column.

Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 9-1. TurboSRX Display Types

TurboSRX [FB1][1]	HP 98730	High-Res Color
TurboSRX [FB2]	HP 98730	High-Res Color
TurboSRX [FB3]	HP 98730	High-Res Color

1 This system comes standard with four overlay planes and one bank of eight image planes, with optional banks of eight planes each.

The supported server modes are Combined, Overlay, Stacked Screen.

Table 9-2. Windows in Overlay Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	doublebuffer?	Xlib	Xlib	SGL	
3	No	PseudoColor	●	●	one color reserved for transparency in combined mode
4	No	PseudoColor	●	●	one color reserved for transparency in combined mode

The supported server modes are Combined, Image and Stacked Screen.

Table 9-3. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	
16	No				Not supported
	Yes (8/8)	PseudoColor	●		only on FB2, FB3
24	No	DirectColor	●		only on FB3
	Yes (12/12)	DirectColor	●		only on FB3

X11 Cursors and Starbase Echos

The following list shows default positions where the Starbase echo and X11 cursor (called echo and cursor, respectively) reside for each of the X11 server operating modes.

HP 98730 Display

- Overlay Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

  If image planes are opened and X11 uses three overlay planes, vector echo resides in cursor plane.

  If image planes are opened and X11 uses four overlay planes, vector echo resides in image planes.

  X11 cursor uses hardware cursor.

- Image Mode

  If image-plane X11 window is opened, raster echo resides in image planes and vector echo resides in cursor plane.

  X11 cursor uses hardware cursor.

- Stacked Screen Mode

  If image-plane X11 window is opened, echo shares three or four overlay planes.

  If image-plane X11 window is opened, raster echo resides in image planes.

  If X11 uses three overlay planes and image planes are opened, vector echo resides in cursor plane.

  If X11 uses four overlay planes and image planes are opened, vector echo resides in image planes.

  X11 cursor uses hardware cursor.

- Combined Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

  If image-plane X11 window is opened, raster echo resides in image planes.

  If image-plane X11 window is opened and X11 uses three overlay planes, vector echo resides in cursor plane.

  If image-plane X11 window is opened and X11 uses four overlay planes, vector echo resides in overlay planes.

  X11 cursor uses hardware cursor.

HP 98731 Display

The `hp98731` driver cannot open an X11 overlay-plane window.

- Overlay Mode

  If image planes are opened and X11 uses three overlay planes, echo resides in cursor plane.

  If image planes are opened and X11 uses four overlay planes, echo is not supported.

  X11 cursor uses hardware cursor.

- Image Mode

  If image-plane X11 window is opened, echo resides in cursor plane.

  X11 cursor uses hardware cursor.

- Stacked Screen Mode

  If image-planes are opened and X11 uses three overlay planes, echo resides in cursor plane.

  If image-planes are opened and X11 uses four overlay planes, echo is not supported.

  X11 cursor uses hardware cursor.

- Combined Mode

  If image-plane X11 window is opened and X11 uses three overlay planes, echo resides in cursor plane.

  If image-plane X11 window is opened and X11 uses four overlay planes, echo resides in overlay planes.

  Raster echo is not supported.

  X11 cursor uses hardware cursor.

Cursors

The HP 98731 Device Driver implements cursors using either the hardware cursor or overlayed software cursors.

Overlayed Software Cursors

If no processes have opened all four overlay planes, then the fourth overlay plane is used for overlayed software cursors either by the HP 98730 or the HP 98731 drivers running in the image planes. The HP 98730 driver running in the overlay planes never uses the fourth overlay plane for cursors. Instead, either the hardware cursor or all three (four) overlay planes are used for cursors.

You can think of the fourth overlay plane used for cursors as a separate "cursor plane". Any data in the cursor plane will be displayed *over* data in the image planes. Data in the other three overlay planes will be displayed *over* data in the image planes and the cursor plane. For example, suppose a graphics application is running in the image planes while the window manager is running in three of the overlay planes. If the application has a Starbase cursor in the overlay cursor plane, the cursor will always be visible inside regions of see-thru color because the cursor has display priority over the graphics. If the cursor is moved outside of regions of see-thru color, it is not visible since the non-see-thru regions in the overlay planes have display priority over the cursor plane.

The TurboSRX also supports a hardware cursor that supports all Starbase echo types. The hardware cursor is drawn to a fifth and sixth overlay plane accessible only by the hardware cursor. There is only one hardware cursor available. Usage of the hardware cursor is defined as follows:

1. If an application is running in a Starbase environment only (that is, the X Window system is not running), the hardware cursor is given to the first process that attempts to use cursors.
2. The X Window system sprite always uses the hardware cursor.
3. Via the `gescape` `R_ECHO_CONTROL`, there is a mechanism for you to control usage of the hardware cursor. This `gescape` is discussed in the appendix.

If the hardware cursor is already being used by another process, then software cursors are used by the HP 98730 or HP 98731 drivers.

If the fourth overlay plane is not available for cursors, an error will be generated when any attempts are made to turn on the cursors. In an X window, cursors may be available even when the fourth overlay plane is not. See *Starbase Graphics Techniques* for more information.

You can control if the software cursors are overlayed in the fourth overlay plane or reside in the same planes currently being used for graphics by the `gescape` `R_OVERLAY_ECHO`. Refer to the appendix for a discussion of this `gescape`.

If your application never uses cursors, the driver will never attempt to allocate the hardware cursor. However, once the driver has allocated the hardware cursor, the driver does not relinquish usage of the hardware cursor until `gclose` time.

If allocation of the hardware cursor was not successful, resources for the software cursor area are allocated (that is, offscreen areas for raster echo definitions). Once resources for software cursors have been allocated, the driver always uses software cursors and never again attempts to use the hardware cursor.

The following functions will cause the driver to attempt to allocate cursor resources (that is, either the hardware cursor or software cursor resources):

- `echo_type` or `define_raster_echo`.
- any of the `gescapes` `R_DEF_ECHO_TRANS`, `R_ECHO_MASK`, `R_ECHO_FG_BG_COLORS`, and `R_OV_ECHO_COLORS`.

The Hardware Cursor

The HP 98731 color map supports a single, independent hardware raster or vector cursor. The hardware cursor is a 64×64×2 bit raster pattern that is conceptually in front of the overlay planes. It is defined with a 64×64 bit/pixel color pattern and a 64×64 bit/pixel transparency pattern. When the X11 server is started, it uses the hardware cursor for the window cursor.

As with the overlay planes, one of the colors is a transparency color used to see through to the overlay and image planes. This means that a raster cursor can have no more then two significant colors (one additional color is used for the transparency pattern). The two colors used by the cursor are based on 24-bit RGB values and are independent of the other color maps.

When the X11 server is using the hardware cursor and a program defines a Starbase echo in an image window, the echo is placed by default in the cursor plane. When a cursor plane is not available, the HP 98730 driver renders the cursor in the image planes. The echo colors will be chosen from the color map associated with that window. When it is an image plane window, the X standard color map is used. This means that when an image plane window is the focused window, the X standard color map will be loaded into the overlay plane hardware color map.

Z-Buffer

For graphics operations that require a Z-buffer such as hidden-surface removal, a dedicated Z-buffer board must be installed in the HP 98731. When the Z-buffer board is installed and an accelerated image-plane X11 window is opened, the X11 server also associates a corresponding portion of the Z-buffer with the window. This Z-buffer allocation is automatically moved and resized as the window is moved and resized. It is also obscured by other windows in the image planes.

Opening Windows

The HP 98731 accelerated driver can open a number of windows in the image planes. The limits placed on these windows are:

1. The HP 98731 driver supports up to 31 accelerated windows operating simultaneously. Furthermore, it permits an accelerated window to be obscured by, at most, 31 other rectangles (for example, corners of windows).
2. When an image plane window is rendered to by the accelerator and is obscured by more than 31 rectangles, rendering is halted until that window has moved up enough in the window stack to be obscured by fewer than 31 rectangles. It is possible for a program to detect when this occurs by passing a procedure address to the Starbase `gescape` procedure with opcode `CLIP_OVERFLOW`. This procedure is then called whenever the clip list overflows.
3. When a window is about to become obscured by more than 31 windows and the accelerator hardware is currently rendering to that window, the window system is locked until the accelerator is finished with the current set of primitives. The calling process will become blocked and the `CLIP_OVERFLOW` procedure will be called by Starbase.

The above guidelines only apply to windows in the image planes. For example, in combined mode, overlay plane windows which overlap image plane windows do not count against the limit of 31 obscuring rectangles. The limit only applies to image-plane windows which overlap other image-plane windows. We recommend that non-graphical windows (for example, terminal emulator windows) and graphical windows that don't need to use the graphics accelerator be placed in the overlay planes.

Note that accelerated overlay windows are not supported with the HP 98731 driver.

Setting Up the Device On Series 300

The HP 98730 and HP 98731 Device Drivers can be used with the graphics display configured in either internal or external DIO-I address space, or in DIO-II address space. Refer to the *Configuration Reference Manual* for a description of internal and external DIO-I address space and DIO-II address space.

Note If the HP 98730 is configured as an external display, there will *not* be an Internal Terminal Emulator (ITE) for that device. Since it is the ITE that normally initializes the display, external devices must be reset after power-up by running a simple Starbase program with a mode of `RESET_DEVICE` in the `gopen` call. It may also be necessary to run this program after running an application which manipulated the overlay color map, such as a windows application program. An example program which could be called from `/etc/rc` during power-up is given at the end of this section. For more details concerning the effects of `RESET_DEVICE`, see the "Device Initialization" information in this section.

The Graphics Interface card may be installed in any DIO slot in the computer's backplane or in any I/O slot of the expander.

DIO-I Switch Settings

The graphics interface card has a single 8-bit address select switch. Looking at the switches so that the dot is in the lower left corner, the leftmost switch is labeled DIO1 to the bottom (0), and DIO2 to the top (1). To configure the system in DIO-I space, this switch must be set to the DIO1 (0) position. The next switch to the right is labeled INT and determines if the HP 98730 workstation is configured as an internal or external display. In addition, the next six switches to the right are labeled for select code determination (five of the six switches are actually used for the select code). There is also a jumper labeled JP1.

The frame buffer uses two megabytes of I/O address space, starting at FB_BASE. The jumper (JP1) determines the address of FB_BASE.

JP1 is set to $200K	FB_BASE address is $200 000
JP1 is set to $800K	FB_BASE address is $800 000

Systems which use the HP 98730 display as a DIO-I system console will map the frame buffer to $200 000; systems which use the display as an external DIO-I device will map the frame buffer to $800 000.

The control space requires 128 Kbytes of space, starting at CTL_BASE. The six switches labeled SC determines the address of CTL_BASE. The HP 98730 may be configured as an external display or as an internal display. Since only 64 Kbytes of space is normally allotted for external I/O select codes, two consecutive select codes will be used when configuring the device as an external display.

The following table lists the binary switch settings with the corresponding values of CTL_BASE for external I/O settings. The table also lists the select codes that are used for each setting.

Table 9-4. DIO-I Control Space Settings (External I/O)

Switch Setting MSB to LSB	CTL_BASE	DIO-I Select Code
01101010	$6A0000	10–11
01101100	$6C0000	12–13
01101110	$6E0000	14–15
01110000	$700000	16–17
01110010	$720000	18–19
01110100	$740000	20–21
01110110	$760000	22–23
01111000	$780000	24–25
01111010	$7A0000	26–27
01111100	$7C0000	28–29
01111110	$7E0000	30–31

For a system console (internal) the switch setting is 01010110 and the CTL_BASE is $560 000.

If the HP 98730 is configured as the system console, the CTL_BASE needs to be placed at $560 000 and the JP1 must be open (no jumper—or jumper is on one pin), which is an interal I/O setting. If the device is not used as the system console, then the control space should not be placed in internal I/O space. It is likely to overlap the address space of other system hardware. In this case, an external I/O space setting should be selected with two consecutive select codes which are not used by the system.

DIO-II Switch Settings

If the left-most switch is set to DIO2 (1), the HP 98730 device can be used in DIO-II address space. In this mode, the next seven switches determine the DIO-II select codes to be used. An HP 98730 device will use three DIO-II select codes. Both the frame buffer and control space reside in the select code areas, so the jumper JP1 is ignored.

The control space requires 4 Mbytes of space, starting at CTL_BASE. The seven switches labeled "SC" at the top of the select switch determine the address of CTL_BASE. The frame buffer requires 8 Mbytes of space, starting at FB_BASE.

Table 9-5. DIO-II Control Space Settings

Switch Setting MSB to LSB	CTL_BASE	DIO-II Select Code	FB_BASE
10000101	$01400000	133	$01800000
10001001	$02400000	137	$02800000
10001101	$03400000	141	$03800000
10010001	$04400000	145	$04800000
10010101	$05400000	149	$05800000
10011001	$06400000	153	$06800000
10011101	$07400000	157	$07800000
10100001	$08400000	161	$08800000
10100101	$09400000	165	$09800000
10101001	$0A400000	169	$0A800000
10101101	$0B400000	173	$0B800000
10110001	$0C400000	177	$0C800000
10110101	$0D400000	181	$0D800000
10111001	$0E400000	185	$0E800000
10111101	$0F400000	189	$0F800000
11000001	$10400000	193	$10800000
11000101	$11400000	197	$11800000
11001001	$12400000	201	$12800000
11001101	$13400000	205	$13800000
11010001	$14400000	209	$14800000
11010101	$15400000	213	$15800000
11011001	$16400000	217	$16800000
11011101	$17400000	221	$17800000
11100001	$18400000	225	$18800000
11100101	$19400000	229	$19800000
11101001	$1A400000	233	$1A800000
11101101	$1B400000	237	$1B800000
11110001	$1C400000	241	$1C800000
11110101	$1D400000	245	$1D800000
11111001	$1E400000	249	$1E800000
11111101	$1F400000	253	$1F800000

DIO-II displays may be used as the system console or as external displays. In order to use the display as system console, it must be configured as the first DIO-II display in the system, and there must be no DIO-I console, or remote terminals. Being the first DIO-II device means that it has the lowest DIO-II select code in the system. In order to use a HP 98730 device as a DIO-II system console, select code 133 is recommended.

Note It is necessary to increase some of the HP-UX tunable system parameters due to the size of the DIO-II mapping of an HP 98730 device. For details on how to reconfigure your kernel, refer to the *HP-UX System Administrator Manual* (particularly the "Configuring HP-UX" section in "The System Administrators Toolbox" and the "System Parameters" appendixes.

It is *essential* that you consult the above referenced HP-UX documentation before you attempt to reconfigure your system. It is possible to adversely affect your HP-UX system if a mistake is made. Ensure you have an understanding of these procedures before proceding.

Example Program to Reset the TurboSRX

The following example uses the HP 98730 device driver. The HP 98731 device driver can be substituted.

```
/*
 * Starbase program: reset98730.c
 * Compile: cc -o reset98730 reset98730.c -ldd98730 -lsb1 -lsb2 -lm
 * Destination: /usr/bin
 * Execute: add line to the /etc/rc - "/usr/bin/reset98730 /dev/crt.external"
 *
 * Example program to be put in /etc/rc for resetting an external HP 98730
 *  device during power-up.
 */
 #include <starbase.c.h>

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

    if ((fildes = gopen(argv[1],OUTDEV,"hp98730",INIT|RESET_DEVICE)) < 0)
       printf("External HP 98730 %s initialization failed.\n",argv[1]);
    else {
       printf("External HP 98730 %s initialization succeeded.\n",argv[1]);
       gclose(fildes);
    }
 }
```

Address Space Usage On Series 300

The TurboSRX is memory mapped into a processes virtual address space, starting at the value specified by the environment variable `SB_DISPLAY_ADDR`. If this variable is not set, then mapping defaults to `0xB00000`. The control space starts at this address and grows towards larger address values. After the control space comes the frame buffer, then shared memory mapped for Starbase drivers. The size of the address space used for control space and the frame buffer depends on whether the device is used in DIO-I or DIO-II. In DIO-I, control space consumes 128 Kbytes and the frame buffer uses 2 Mbytes. In DIO-II, control space is 4 Mbytes and the frame buffer is 8 Mbytes. The size of the Starbase drivers' shared memory is always the same, and is slightly less than 300 Kbytes.

If your application maps memory pages to specific addresses, or needs a large stack, then you may need to adjust SB_DISPLAY_ADDR to avoid conflicts.

Special Device Files (mknod) On Series 300

The mknod command creates a special device file which is used to communicate between the computer and the peripheral device. See the mknod(1M) information in the *HP-UX Reference* for further information. The name of this special device file is passed to Starbase in the gopen procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the /dev directory. Any name may be used for the special device file, however the name that is suggested for these devices is crt.

The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the mknod command.

When the device is at the internal DIO-I address (refer to the "Switch Settings" section) the mknodh parameters should create a character special device with a major number of 12 and a minor number of 0. Note that the leading 0x causes the number to be interpreted hexadecimally.

```
mknod /dev/crt c 12 0x000000
```

When the device is at an external DIO-I or any DIO-II address (refer to the "Switch Settings" section) the mknod parameters should create a character special device with a major number of 12 and a minor number of 0x⟨*sc*⟩0200 where ⟨*sc*⟩ is the two-digit external select code in hexadecimal notation.

```
mknod /dev/crt c 12 0x⟨sc⟩0200
```

The HP 98730 and HP 98731 Device Drivers may also be used for the overlay planes in graphics mode. The minor number may be set to cause Starbase drivers to use either three or four overlay planes. When running to three planes, one plane is still reserved for cursors. When running to all four overlays, only the hardware cursor is available for Starbase graphics echoes. If more than one echo is requested, or if another process is using the cursor, the request for another echo will fail. Note that since the terminal emulator and window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is opened in this manner while these processes are present. To open either the HP 98730 or HP 98731 Device Drivers to three overlay planes instead of the graphics planes, the last byte of the minor number must be one. To run to all four overlays, the last byte of the minor number must be three.

For example, when the device is at an internal DIO-I address, the **mknod** parameters for the overlay device, with one plane reserved for cursors, should create a character special device with a major number of 12 and a minor number of 1.

```
mknod /dev/ocrt c 12 0x000001
```

To create a device file for all four overlays, the command would be:

```
mknod /dev/o4crt c 12 0x000003
```

When the device is at an external DIO-I address or any DIO-II address (refer to the section on "Switch Settings") the **mknod** parameters for the same device should create a character special device with a major number of 12 and a minor number of `0x`⟨*sc*⟩`0201` or `0x`⟨*sc*⟩`0203` where ⟨*sc*⟩ is the two-digit select code.

```
mknod /dev/o3crt c 12 0x⟨sc⟩0201
```

or

```
mknod/dev/o4crt c 12 0x⟨sc⟩0203
```

Setting Up the Device on the Series 800

Up to four HP 98730 or HP 98731 devices can be connected to a Series 800 SPU using four A1017A interface cards. However, it is recommended that only two HP 98730 devices or HP 98731 devices have the Internal Terminal Emulator (ITE) or window systems running on them. With the A1047A interface, only two devices can be connected, both of which may run an ITE.

The Series 800 ITE supports power-fail recovery on the HP 98730 device or HP 98731 device, but Starbase does not support this feature. If you want to support power fail, you must catch the power-fail signal and save any Starbase state needed. Then, `gclose` the device and `gopen` the device again when the power turns on.

Special Device Files (mknod) On the Series 800

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod`(1M) information in the *HP-UX Reference* for further details. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however, the names that are suggested for the devices are `crt`, `crt0`, `crt1`, or `crt2`.

The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the `mknod` command.

When creating the device file the `mknod` parameters should create a character special device with a major number of 14 and a minor number of the format below (where ⟨*lu*⟩ is the two-digit hardware logical unit number):

```
mknod /dev/crtx c 14 0x00⟨lu⟩00
```

9

The HP 98730 Device Driver may also be opened to the overlay planes in graphics mode. If the last byte of the minor number is one, 3-overlay planes are used for graphics (and the fourth plane is reserved for cursors for processes running in the image planes). If the last byte of the minor number is three, 4-overlay planes are used for graphics. Since the ITE and window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is open in this manner while these processes are present.

To open all 4-overlay planes when the device is at the internal address, the **mknod** parameters should create a character special device with a major number of 14 and a minor number of three.

```
mknod /dev/o4crt c 14 0x00⟨lu⟩03
```

To open three overlay planes when the device is at the internal address, the **mknod** parameters should create a character special device with a major number of 14 and a minor number of one.

For example, the **mknod** parameters for a 3-plane overlay device should create a character special device with a major number of the format indicated below (where ⟨*lu*⟩ is the hardware logical unit number):

```
mknod /dev/o3crt c 14 0x00⟨lu⟩01
```

Linking the Driver

Shared Libraries

The shared HP 98730 Device Driver is the file named `libdd98730.sl` in the `/usr/lib` directory. The shared HP 98731 Device Driver is the file named `libdd98731.sl` in the `/usr/lib` directory. The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Upon device initialization the proper driver will be loaded. See the discussion of the `gopen` procedure in the **Device Initialization** section of this chapter for details.

Archive Libraries

The HP 98730 Device Driver is located in the `/usr/lib` directory with the file name `libdd98730.a`. The HP 98731 Device Driver is located in the `/usr/lib` directory with the file name `libdd98731.a`. The device driver may be linked to a program using the absolute path name, for example `/usr/lib/libdd98730.a`, or an appropriate relative path name, or by using the `-l` option as in `-ldd98730` with the `LDOPTS` environmental variable set to `-a archive`.

The reason for using the `LDOPTS` environmental variable is that the `-l` option will look for a shared library driver first and then look for the archive driver if shared was not found. By exporting the `LDOPTS` variable as specified above, the `-l` option will only look for archive drivers. For more information, refer to the *Programming on HP-UX* manual on linking shared or archive libraries.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with the HP 98730 driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldd98730 -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldd98730 -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldd98730 -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Usage and Restrictions

HP 98730

For the HP 98730 device driver, when a device file for the overlay planes is used at gopen time, bank switching is not supported.

Graphics applications that want to talk to a graphics window may use this device driver. If the graphics window is in the overlay planes, this device driver does not support:

- Bank switching.
- Z-buffering.
- Double buffering when using three overlay planes. (Double buffering in a window in the overlay planes is supported to 4-overlay planes).
- Shading.
- The transform engine.

If the graphics window is in the image planes, this device driver does not support:

- Z-buffering.
- Shading.
- The transform engine.

Graphics applications that want to talk to a local X window can use this device driver. If the window is in the overlay planes, this device driver does not support:

- Bank switching.
- Z-buffering.
- Double buffering.
- Shading.
- The transform engine.

If the graphics window is in the image planes, this device driver does not support:

- Z-buffering.
- Shading.
- The transform engine.

Transparency Index

There are four overlay planes in the TurboSRX. Even though these planes can display 16 colors simultaneously, only 15 are available because one color is reserved for the transparency color. By default, this color is index 7 or 15, depending on the overlay depth. When the transparency color's index is written into the overlay planes, the observed color is that of the image planes. The transparency color is set when the X11 server is started and cannot be changed until the server is shut down.

The see-thru facility allows you to create a transparent window.

By default index 3 (yellow) is reserved as see-thru. The HP 98730 Device Driver recognizes the Starbase environment variable `SB_OV_SEE_THRU_INDEX` that will allow you to set the see-thru color map index to some other value. This environment variable will only have effect when running the program on the `raw` device.

When running the `raw` device, an explicit call to `define_color_table` will cause see-thru entries to be set back to `dominant`. When running to an overlay graphics window, an explicit call to `define_color_table` will preserve the see-thru entry currently defined to the window system.

HP 98731

When a device file for the overlay planes is used at `gopen` time, bank switching, shading, and depth cueing are not supported.

Graphics applications that want to open a local X window may use this driver if the window is in the image planes.

Up to 32 HP 98731 device drivers may be opened to the same device simultaneously from any combination of one or more processes.

X Window System See_Thru Color

The X Window system always uses color 7 (15 for 4-plane devices) as the see-thru color. This cannot be changed.

Device Initialization

Parameters for gopen

The `gopen` procedure has four parameters: Path, Kind, Driver, and Mode.

Path	The name of the special device file created by the `mknod` command as specified in the last section, e.g. `/dev/crt`.
Kind	Indicates the I/O characteristics of the device. This parameter may be one of the following: ■ `INDEV`, Input only. ■ `OUTDEV`, Output only. ■ `OUTINDEV`, Input and Output. Input may be done with this driver only when opened to an X Window system window.
Driver	The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire the device and by default load the accelerated driver (if applicable). For example: `NULL` for C `char(0)` for FORTRAN77 '' for Pascal Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is ignored. For example: `"hp98730"` *for C.* `'hp98730'//char(0)` *for Fortran77.* `'hp98730'` *for Pascal.*
Mode	The mode control word consisting of several flag bits *or* ed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure.

- SPOOLED, cannot spool raster devices.
- MODEL_XFORM—Shading is not supported for the HP 98730 device. However, opening in MODEL_XFORM mode will affect how matrix stack and transformation routines are performed.
- 0—Open the device without clearing the screen. This will set the color map mode to CMAP_NORMAL, but will not initialize the color map itself. This will also disable blending if it was left enabled. This mode will not affect pixel panning and zooming.

  In an X Window the color map mode is initialized consistent with the X color map.
- INIT—open and initialize the device as follows:
  1. Clear frame buffer to 0s.
  2. Reset the color map to its default values.
  3. Enable the display for reading and writing.
  4. HP 98731 - Initialize the transform engine's microcode.
  5. HP 98731 - Download the transform engine's microcode (if it has not already been done).
  6. Restore pixel pan and zoom hardware for normal viewing, if opened to the image planes.
  7. In an X Window a new color map is created and the color map mode is initialized to be consistent with the X color map.
- RESET_DEVICE—open and reset the device as follows:
  1. Clear frame buffer and overlays to 0s.
  2. Reset the color map to its default values.
  3. Clear the overlay color map.
  4. Enable the display for reading and writing.
  5. HP 98731 - Download the transform engine's microcode.
  6. HP 98731 - Initialize the transform engine's microcode.
  7. Restore pixel pan and zoom hardware for normal viewing, if opened to the image planes.
  8. HP 98731 - In an X window, a new color map is created, and the color map mode is initialized to be consistent with the X color map.
  9. Reset the graphics accelerator.

Note that the `RESET_DEVICE` flag bit should be used with caution: it will adversely affect any other processes using the device. This flag bit is intended to reset a device completely: this should only be necessary for devices in an unknown state, such as a device powered up in an external I/O space. Most programs should not use this flag bit.

Syntax Examples

To open and initialize a TurboSRX for output:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV,NULL,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,'',INIT);
```

Special Device Characteristics

For Device Coordinate operations, location (0, 0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is therefore (1279, 1023).

Offscreen Memory Usage

Offscreen memory is managed by a global resource manager to insure that multiple processes do not step on each other when using the offscreen. Offscreen is used by the device driver for:

- polygon fill patterns
- raster fonts
- raster echo definitions (if software cursors are used)

The offscreen memory is not allocated for any of the above functions unless the function is used. Therefore, if an application never does filled polygons, never uses software cursors, and never uses raster fonts; the driver does not use the offscreen memory. Refer to the `gescape R_OFFSCREEN_ALLOC` for information on using the offscreen areas for personal use.

Device Defaults

Number of Color Planes

When the `gopen` procedure is called, this driver asks the device for the number of color planes available. This number can be either 3 or 4 (if running to the overlay planes), 8, 16, or 24. The device driver then acts accordingly.

Dither Default

The default number of colors searched for in a dither cell is 2. The number of colors allowed in a dither cell is 1, 2, 4, 8 or 16. For devices having 24 or more planes in `CMAP_FULL` mode (see `shade_mode`) dithering is not supported since full 24-bit color is available. If you are double buffering with 12 planes per buffer then the number of colors allowed in a dither cell is 1, 2, or 4.

Raster Echo Default

The default raster echo is the 8×8 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 allowed for a raster echo is 64×64 pixels. The default drawing mode for the raster echo is 7 (*or*).

If the driver does not have access to the hardware cursor, by default the raster echo is written to the same planes currently being used for graphics. For example, if the HP 98730 driver was opened to the image planes, the image planes are used for raster cursors. If the HP 98730 driver was opened to three overlay planes, the those three overlay planes are used for raster cursors. The location of software raster and software non-raster cursors can be changed using the `gescape` function `R_OVERLAY_ECHO`.

Color Planes Defaults

In a raw display, the default configuration is an 8-plane color mapped system regardless of the number of frame buffer banks installed. In an X Window the color plane definition is consistent with the X color map.

All planes in first bank are display enabled. All planes in first bank are write enabled.

Semaphore Default

Semaphore operations are enabled.

Line Type Defaults

The default line types are created with the bit patterns shown below:

Table 9-6.

Line Type	Pattern
0	1111111111111111
1	1111111100000000
2	1010101010101010
3	1111111111111010
4	1111111111101010
5	1111111111100000
6	1111111111110110
7	1111111110110110

Default Color Map

If the fourth `gopen` parameter is zero (`0`), the current hardware color map is used on raw color displays. On X Windows the window's current color map is used.

If the fourth `gopen` parameter is `INIT`, the current color map is initialized to the default values shown in the following table.

Table 9-7. 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

Use the `inquire_color_map` procedure to see the rest of the 255 colors.

When `INIT` is used in the `shade_mode` procedure call the color map will be initialized dependent on the `mode` parameter and the number of frame buffer banks installed.

`CMAP_NORMAL` Only one bank of the three banks can be displayed at a time, unless video blending is enabled.

CMAP_MONOTONIC The color map will be initialized as:

```
          for (i=0; i<256; i++) {
             cmap[i].red = cmap[i].green = cmap[i].blue = i/255.0;
        }
```

Only one bank of the three banks can be displayed at a time, unless video blending is enabled.

CMAP_FULL With less than three banks installed the color map will be initialized as three bits red, three bits green and two bits blue. The three most significant bits are red and the two least significant bits are blue. Only one bank of the three banks can be displayed at a time.

With three or more banks installed the color map will be initialized as the CMAP_MONOTONIC case above, the first bank of eight will go through the blue portion of the color map, the second bank goes through the green portion and the third bank goes through the red portion. In this mode the color map is transparent and the eight bits from each bank drives the appropriate DAC. The color map could be subsequently modified in this mode to do things like gamma correction or double buffering of four bits per color.

Red, Green and Blue

Each file descriptor opened as an output device has a color table associated with it. If multiple file descriptors are open to the same device, the color table and the device's color map may not always be identical. The color table does not track the color map if the device's color map is changed by another file descriptor path.

For Starbase procedures that have parameters for red, green and blue, the way the actual color is chosen depends on the current shade_mode setting.

CMAP_NORMAL The color map is searched for the color that is closest in RGB space to the one requested, and that color map index is written to the frame buffer for subsequent output primitives. It is more efficient to select a color with an index rather than specifying a color with red, blue and green values in this mode due to the time it takes to figure out which index in the color table most closely matches the specified color.

CMAP_MONOTONIC	The red, green and blue value is converted to an intensity value using the equation: `0.30*red+0.59*green+0.11*blue` This intensity is converted to an index value by mapping intensity 0.0 to the minimum index set by `shade_range` and intensity 1.0 to the maximum index set by `shade_range`. This mode is useful for displaying a high quality monochrome picture on an 8-plane system from data that produces a high quality color picture on a 24-plane system.
CMAP_FULL	The color values will be mapped directly to an index with the assumption the color map is setup to a predefined full color state.

Note Multiple `gopen` parameters of an X Window will share a single color map definition. See the *Starbase Graphics Techniques* for more information.

Starbase Functionality

Commands Not Supported

The following commands are not supported by the HP 98730 and HP 98731 device drivers. If one of these commands is used by mistake, it will be ignored and not cause an error.

```
alpha_transparency          line_filter
bf_alpha_transparency       perimeter_filter
bf_texture-index            set_capping_planes
contour_enable              set_model_clip_indicator
define_contour_table        texture_index
define_texture              texture_viewport
deformantion_mode           tecture_window
```

The following commands are not supported by the HP 98730 device driver:

```
backface_control            light_ambient
bf_control                  light_attenuation
bf_fill_color               light_model
bf_interior_style           light_source
bf_perimeter_color          light_switch
bf_perimeter_repeat_length  set_model_clip_volume
bf_perimeter_type           shade_range
bf_surface_coefficients     surface_coefficients
bf_surface_model            surface_model
define_trimming_curve       viewpoint
depth_cue                   zbuffer_switch
depth_cue_color
depth_cue_range
hidden_surface
```

Commands Conditionally Supported

HP 98730 Device Driver

The following commands are supported on the HP 98730 device driver under the listed conditions:

`block_read, block write`	The `raw` parameter for the `block_read` and `block_write` commands is normally ignored by this driver. To use the `raw` mode, you must call the `R_BIT_MODE gescape` discussed in the appendix of this manual.
`pattern_define`	4×4 is the largest supported pattern.
`shade_mode`	The color map mode may be selected but shading can not be turned on.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	The `use` parameter must be zero, any extra coordinates supplied will be ignored.
`with_data`	`partial_polygon_with_data3d` `polygon_with_data3d` `polyhedron_with_data` `polyline_with_data3d` `polymarker_with_data3d` `quadrilateral_mesh_with_data` `triangle_strip_with-data` Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

HP 98731 Device Driver

The following commands are supported on the HP 98731 device driver under the listed conditions:

`inquire_fb_configuration`	An HP 98730 device running the HP 98731 Device Driver will report `image_banks` as 5 if the system has 24 display planes and the dedicated Z-buffer. If the dedicated Z-buffer is installed in this way, it is possible to access it with `block_write`, `block_read`, and `block_move`. The Z-buffer may be selected for read/write using `bank_switch`. The Z-buffer may not be displayed. The Z-buffer cannot be rendered to by the graphics accelerator. If less than 24 planes are installed, the presence of a Z-buffer will not be reported.
`interior_style`	If the polygon fill type is `INT_HATCH` then the following functionality will not work correctly: ■ hidden surface removal ■ shading and lighting ■ depth cueing ■ backfacing attributes and culling ■ splines, quadrilateral meshes, and triangle strips will not be hatched. Performance is also degraded in this mode.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`with_data`	`partial_polygon_with_data3d` `polygon_with_data3d` `polyhedron_with_data` `polyline_with_data3d` `polymarker_with_data3d` `quadrilateral_mesh_with_data`

`triangle_strip_with-data`

Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

block_read, block_write (HP 98731)

The `raw` parameter for the `block_read` and `block_write` commands is normally ignored by this device driver. To use the `raw` mode, you must call the `R_BIT_MODE` or `R_DMA_MODE` `gescapes` discussed in the appendix of this manual. If the `raw` parameter is `TRUE`, then no clipping will be done.

When running to a window, the window offsets from the upper left hand corner of the screen will be added to `block_write` and `block_read` start locations. If you do not want this offset added, you should subtract the offsets from your start point. These offsets can be computed by calling the `gescape` functions `R_GET_FRAME_BUFFER` and `R_GET_WINDOW_INFO`. Using the frame buffer pointers returned by these routines, the window offsets are:

```
y_offset=(window_ptr-fb_ptr)/2048

x_offset=(window_ptr-fb_ptr)-(y_offset*2048)
```

This would be useful, for example, if you wished to write a polygon fill pattern offscreen to a frame buffer absolute address while running in a window.

Fast Alpha and Font Manager Functionality

The HP 98730 Device Driver supports raster text calls from the fast alpha and font manager libraries. These calls may be made while running in the overlay or image planes. Since raster fonts consist of one byte per pixel, image plane raster text is written only to the currently selected bank. This is similar to the operation of other raster functions such as `block_write`. Fast alpha and font manager fonts can be optimized. See the *Fast Alpha/Font Manager Programmer's Manual* for further information.

The HP 98731 Device Driver does *not* support raster text calls from the fast alpha and font manager library.

Parameters for gescape

The HP 98730 and HP 98731 support the following gescape operations. Refer to Appendix A of this manual for details on gescapes.

- BLINK_INDEX—alternate between HP 98730 hardware color maps. This gescape is not supported while image blending is active. Refer to the IMAGE_BLEND gescape.
- BLINK_PLANES—blink the display (blink rate is 3.75 Hz for this device)
- IMAGE_BLEND—control analog blending of image plane frame buffer output
- OVERLAY_BLEND—control analog blending of overlay plane frame buffer output
- PAN_AND_ZOOM—do pixel panning and zooming
- R_BIT_MASK—bit mask
- R_BIT_MODE—bit mode
- R_DEF_ECHO_TRANS—define raster echo transparency
- R_DEF_FILL_PAT—define fill pattern
- R_DMA_MODE—changes definition of raw for block writes
- R_ECHO_CONTROL—control hardware cursor allocation
- R_ECHO_FG_BG_COLORS—define cursor color attributes
- R_ECHO_MASK—define a raster echo mask pattern
- R_FULL_FRAME_BUFFER—full frame buffer
- R_GET_FRAME_BUFFER—read frame buffer address
- R_GET_WINDOW_INFO—returns frame buffer address of window
- R_LINE_TYPE—define line style and repeat Length
- R_LOCK_DEVICE—lock device
- R_OFFSCREEN_ALLOC—allocates offscreen frame buffer memory
- R_OFFSCREEN_FREE—frees allocated offscreen frame buffer memory
- R_OV_ECHO_COLORS—select overlay echo colors
- R_OVERLAY_ECHO—select plane to contain cursor
- R_TRANSPARENCY_INDEX—specify HP 98730 transparency index
- R_UNLOCK_DEVICE—unlock device
- READ_COLOR_MAP—read color map
- SET_BANK_CMAP—define bank color map to be used for image blending
- SWITCH_SEMAPHORE—semaphore control

The HP 98730 also supports the following gescape:

- R_OVERLAY_ECHO—select plane to contain cursor.

The HP 98731 also supports the following gescapes:

- CLIP_OVERFLOW—Change X Window system hierarchy.
- GAMMA_CORRECTION—Enable/disable gamma correction.
- PAN_AND_ZOOM—Pixel pan and zoom.
- PATTERN_FILL—Fill polygon with stored pattern.
- POLYGON_TRANSPARENCY—Define front facing and backfacing polygon transparency patterns.
- TRANSPARENCY—Allows "screen door" for transparency pattern.
- ZWRITE_ENABLE—Allows creation of 3D cursors in overlay.

Performance Tips

HP 98730 and HP 98731 Device Drivers

1. As with any driver, buffering is done to enhance performance. Performance can be degraded if `buffer_mode` is turned off or an inordinate amount of `make_picture_current` calls are done.
2. Performance optimizations have been made so that sequential calls of the same output primitive, with no intervening attribute changes or different primitive calls, go faster. For example the sequence `line_color`, `polyline`, `polyline` is faster than `line_color`, `polyline`, `line_color`, `polyline`. So grouping by primitive and subgrouping primitives by attribute can give some performance improvements.

HP 98730 Device Driver

1. If only one process is accessing the graphics display, it is safe to turn off the semaphore operations (see the `SWITCH_SEMAPHORE gescape`), and a 10 to 20 percent speed improvement can be obtained. If a tracking process is initiated, then semaphores will automatically be turned on.
2. If Starbase echos are overlayed (i.e. in the fourth overlay plane), or hardware cursors are used, graphics performance is significantly better since it is not necessary to "pick up" the cursor each time the frame buffer is updated.
3. Screen clears will be significantly faster if the area to be cleared starts on a 128-pixel boundary and is some multiple of 128-pixels wide. This can be checked by using the Starbase routines `transform_point` and `vdc_to_dc` to convert the bounds of the clear rectangle to device coordinates. Screen clears to the default `vdc_extent` will be aligned. Screen clears are also much faster when the background color index is zero. Screen clears with a non-zero index require two pases, which result in slower performance.
4. Polygons are filled faster when the drawing mode is ⟨*SOURCE*⟩, `NOT_SOURCE`, `ZERO`, or `ONE`.
5. Horizontal and vertical lines are faster than diagonal lines on this device since the hardware block mover is used to generate pixels.

6. The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.
7. Performance of `block_read` and `block_write` is significantly better if both the source and destination begin on the same byte boundary (since data can be transferred 32 bits at a time rather than one byte at a time). For example, one way to ensure this condition is to define pixel arrays as type short (16-bit integer), and start `block_read` and `block_write` on even pixels only. This can more than double performance.
8. `block_write` on Series 800 machines with the A1047A interface can go faster by using DMA. See `R_DMA_MODE gescape`.

HP 98731 Device Driver

1. Typically, the HP 98731 rendering engine renders primitives from its internal buffer as the system CPU is doing other things. Substantial performance benefits can be realized from this parallel processing.

   However, certain operations will cause the CPU to wait for the HP 98731 to finish emptying its buffer. An example of this wait is the `make_picture_current` operation. Also, any operation that reads information from the HP 98731 may cause this wait to occur. Two operations read the matrix values from the HP 98731: `pop_matrix2d` and `pop_matrix3d`. If the values in the popped matrix are not needed, use `pop_matrix`, which does not cause any information to be read from the HP 98731. Also, `block_read` and `block_write` will also cause the driver to use it.

Screen Clears

1. Screen clears will be significantly faster if the area to be cleared starts on a 128-pixel boundary and is some multiple of 128 pixels wide. This can be checked by using the Starbase routines `transform_point` and `vdc_to_dc` to convert the bounds of the clear rectangle to device coordinates. Screen clears to the default `vdc_extent` will be aligned.
2. For programs which use hidden surface removal with the dedicated Z-buffer, it is much faster to clear the Z-buffer simultaneously with screen clears than to do the clears sequentially. This is accomplished by calling `clear_control` with `CLEAR_ZBUFFER` *or* ed into the mode word. When this is done, subsequent

calls to `clear_view_surface` and `dbuffer_switch` will cause the zbuffer to be cleared also. See the manual page for `clear_control` for more details.

Rendering

1. When drawing shaded polygons, the fewer the features, the faster the polygon generation. Positional viewpoint and light sources can significantly degrade performance.

2. With shading, and Z-buffering off, the HP 98731 rendering engine runs at full speed, when rendering flat shaded polygons. These two rendering techniques slow the rendering of polygons on the HP 98731. This is especially noticeable on large polygons. Turning on any one of these could noticeably lower the rendering performance.

   Using the pattern `gescape` or replacement rules that require extra reads of the frame buffer (e.g. source *or* destination) will also degrade performance. It takes time to do the extra reads.

3. Rendering mode commands such as `hidden_surface`, `shade_mode, and double_buffer` can be slow. These should not be unnecessarily called. For example, it is not necessary to repeatedly call `hidden_surface` from an animation loop; it is intended that these routines be called to initialize a rendering mode and are only called again to change it.

Raster Operations

1. The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.

2. The performance of `block_read` and `block_write` is significantly better if both the source and destination begin on the same byte boundary, since data can be transferred 32-bits at a time rather than one byte at a time. For example, one way to ensure this condition is to define pixel arrays as type short (16-bit integers) and then start `block_read` and `block_write` actions on even pixels only. This can more than double performance. Note that the byte boundaries are relative to the screen address, not the window address.

3. `block_write` on Series 800 machines with the A1047A interface can go faster by using DMA. See `R_DMA_MODE gescape`.

Cautions

The following caution is provided in using either the HP 98730 or HP 98731 device drivers:

1. As mentioned previously, accessing the off-screen portion of the frame buffer (using `gescape` functions) should be done with care, since other processes access this region. The overlay off-screen contains the ITE font (which is regenerated when control-shift-reset is done on the ITE keyboard) and may contain any number of window systems fonts depending on the current window usage.
2. Certain `gescape` functions should be used with caution since they bypass protection mechanisms used to prevent multiple processes from interferring with each other. For example, since the hardware resources can only be rationally used by one graphics process at a time, the driver activates a semaphore and locks the device before doing any output. This ensures, for example, that process A will not change the replacement rule while process B is in the middle of filling a polygon. It also prevents the terminal (`tty`) driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when done processing output. Some of the `gescapes` listed in this chapter allow you to change this locking mechanism and should be used with *great caution*.

HP 98731 Device Driver

The following cautions are provided in using the HP 98731 device driver:

1. Polygons of up to 255 vertices (after clipping) are supported. If a polygon has more than 255 vertices, only the first 255 vertices are displayed.
2. When using the TurboSRX with a graphics accelerator it is possible for illegal operations to cause the transform engine or scan converter hardware to enter an unknown state. If this happens, Starbase will report an error the next time it tries to use the hardware. You will see this as a `Transform engine timed out` or `Hardware/scan_converter time out` error. These are Starbase errors 14 and 52 respectively. This is a very serious error condition. If the HP 98731 Device Driver is being used, this is a fatal error. When this error is discovered, Starbase reports the error and aborts execution.

If an application needs to take some emergency action before an untimely termination, such as saving valuable data, the application should check for these error conditions and take appropriate measures. Errors may be caught by an application using the `gerr_control` procedure described in the *Starbase Reference* manual.

It is also possible to avoid the termination completely if the application's error handler does not return control to Starbase. It is, however, impossible to proceed with any graphics efforts using the accelerator until it is reset.

3. The HP 98731 driver does not support rendering off the left-hand or top edge of the display. Therefore, in X11, avoid moving an accelerated starbase window off these edges. In raw mode, make sure that the clip rectangle is set to prevent rendering off the top or left-hand edges.

FINAL TRIM SIZE : 7.5 in x 9.0 in

10

The TurboVRX Device Driver

Device Description

The TurboVRX Graphics Display Station is a graphics subsystem which interfaces with a host SPU, utilizing a high-resolution 16 or 19 inch (1280×1024) color display (purchased separately). The HP 98736 and HP 98766 Display Controllers include a graphics accelerator. The TurboVRX supports the X Window System and Starbase graphics libraries, and can also be used as a system console. The host interface plugs into an I/O slot on the SPU, with a connecting cable to the Display Controller. See the *Introduction* section of this manual for systems supporting this controller and accelerator.

The TurboVRX Graphics Display Station is available in the following configurations:

HP 98735A	■ 24 image plane (frame buffer). ■ 24 planes Z-buffer. ■ 1 Transform Engine.
HP 98736A	■ 24 image plane (frame buffer). ■ 24 planes Z-buffer. ■ 2 Transform Engines.
HP 98736B	■ 24 image plane (frame buffer). ■ 24 planes Z-buffer. ■ 3 Transform Engines.
HP 98765A	■ 24 image plane (frame buffer). ■ 24 planes Z-buffer. ■ 2 Transform Engines.
HP 98766A	■ 24 image plane (frame buffer). ■ 24 planes Z-buffer. ■ 4 Transform Engines.

Double buffering can be enabled for up to 12-bits per buffer. Dithering allows the device to maintain image quality even when double buffering. The 24 bit Z-buffer provides one-pass hidden surface removal.

Series 300 and 400

Two Device Drivers are provided to access the TurboVRX Display Controller which is supported on the Series 300 and Series 400:

- `hp98735`—The HP 98735 Device Driver is used to access the graphics display without using the graphics accelerator.
- `hp98736`—The HP 98736 Device Driver is used to access the graphics display using only the graphics accelerator.

Series 700

Two Device Drivers are provided to access the TurboVRX Display Controller which is supported on the Series 700:

- `hp98765`—The HP 98765 Device Driver is used to access the graphics display without using the graphics accelerator.
- `hp98766`—The HP 98766 Device Driver is used to access the graphics display using only the graphics accelerator.

Note On the Series 700, you may use the `hp98735` and `hp98736` drivers with the TurboVRX Display Controller. These will function the same as the `hp98765` and `hp98766` drivers.

The HP 98736 and HP 98766 display systems are designed to provide accelerated Starbase graphics within the X Window System environment. The display system supports the X Window System by providing window clipping, multiple color maps, and multiple display modes.

The frame buffer subsystem is a bit-mapped device organized as an array of words, with each word representing a pixel on the display. The frame buffer has special hardware for:

- Write enable/disable individual planes.
- Video enable/disable individual planes.
- Memory writes with specified replacement rule.
- Video blinking of individual planes.
- Video blinking of individual color map locations.
- Arbitrary sized rectangular memory to memory copies.

On the HP 98736 and HP 98766 accelerated display stations, the scan conversion subsystem is implemented with custom VLSI to allow high-performance rendering of vectors and polygons, in addition to the following features:

- Gouraud Shading
- Anti-aliasing
- Texture Mapping
- Alpha Transparency

The transform engine subsystem is implemented with parallel floating point processors, each with local storage for instructions and data. The number of engines varies with the configuration, from a minimum of one to a maximum of four transform engines. The transform engines help accelerate a variety of functions:

- Matrix Transformations
- Shading Calculations
- Spline Tesselation
- Model Clipping
- Capping
- Contouring

Refer to the *Starbase Reference Manual* for detailed descriptions of these features.

HP 98735 and HP 98765

Each display is organized as an array of integers, with each integer representing a pixel on the display. In single bank mode, color map indices range from 0 to 255. The color map is a RAM table that has 16, 256, or 4096 addressable locations and is 24 bits wide. Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

In addition to the three frame buffer banks of eight planes each, four overlay planes are provided. These overlay planes have their own unique color map, separate from the color map used for the image planes. This color map consists of sixteen 24 bit entries, allowing the user to select sixteen colors from the full palette of over 16 million choices. In addition, each entry in the overlay color map may be set to be dominant or non-dominant.

When operating with the X window system, the hardware provides 15 separate overlay/image color-map pairs. This allows up to 15 unique software color maps to be simultaneously displayed in their respective windows. This is done using a method that is transparent to the user. For more information on this feature, see the "X Windows" chapter of the *Starbase Graphics Techniques*.

A dominant entry causes all pixels in the overlays set to that value to display the color in the overlay map, regardless of values in the image planes below. A non-dominant entry causes pixels with that value to display the color in the image planes below.

You can use overlay planes for non-destructive alpha, graphics, or cursors. For example, when the `hp98735` driver is used on the system console, the Internal Terminal Emulator (ITE) uses three of the overlay planes for alpha information. This way there is no interaction between ITE text and images in the graphics planes. To do graphics in the overlay planes the `hp98735` device driver may be opened directly to the overlay planes as if they were a separate device. Refer to the section *Setting up the Device* in this chapter for more information.

Typically, you do not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase does provide the `gescape` `R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer (this gescape is discussed in the appendix of this manual). Frame buffer locations are then addressed relative to the returned address. The first word of the frame buffer (word 0) represents the upper left corner pixel of the screen. Word 1 is immediately to its right. Word 1279 is the last (right-most) pixel on the top line. The next 768 words of the frame buffer are not displayable. Word 2048 is the first (left-most) pixel on the second line from the top. The last (lower right corner) pixel on the screen is word number 2,096,383.

If writing to the HP 98735 or HP 98765 image buffer and not in `CMAP_FULL` color map mode, only one bank can be written at a time. The bank to be written must be established by a call to `bank_switch`. When writing the pixel value, the byte position of the value to be written must be appropriate for each bank:

- When writing to bank 0, the pixel value is in the least significant byte of the integer value (byte position 0).
- When writing to bank 1, the pixel value is in byte position 1 of the integer value.
- When writing to bank 2, the pixel value is in byte position 2 of the integer value.

All three banks for one pixel can be written simultaneously by packing all three bank values for the pixel into the integer value and having the color map mode as `CMAP_FULL` before writing.

The off-screen portion of the frame buffer may be accessed via the `gescape`, `R_FULL_FRAME_BUFFER`. See the *Gescape* section in this chapter. Care should be taken when using this `gescape` since other processes, Starbase, and the window system access the frame buffer off-screen memory.

Display modes

The following two tables summarize the supported display modes with the `hp98735`, `hp98765`, `hp98736` and `hp98766` device drivers:

Table 10-1. Display Modes Supported (Single Buffer)

Shade Mode	Display Mode	Color Map Used
CMAP_NORMAL	8 Bit	256 Entry
CMAP_MONOTONIC	8 Bit	256 Entry
CMAP_FULL	8:8:8	3×256 Entry

Table 10-2. Display Modes Supported (Double Buffer)

Shade Mode	Plane/Buffer	Color Map Used
CMAP_NORMAL	1,2,3,4	256 Entry
CMAP_NORMAL	6,8	256 Entry
CMAP_NORMAL	12	4096 Entry
CMAP_MONOTONIC	1,2,3,4	256 Entry
CMAP_MONOTONIC	6,8	256 Entry
CMAP_MONOTONIC	12	4096 Entry
CMAP_FULL	8 (3:3:2)	3×256
CMAP_FULL	12 (4:4:4)	3×256

Aside from support of display modes used on previous devices, a new display mode is supported:

- 12-bit double-buffered indexing in `CMAP_MONOTONIC` and `CMAP_NORMAL` shade modes. (In a single-buffered, image- or combined-mode X server, this display mode is available as a single-buffered mode.)

Note This new display mode is discussed in the *Starbase Graphics Techniques* manual.

FINAL TRIM SIZE : 7.5 in x 9.0 in

HP 98736 and HP 98766 Advanced Features

Texture mapping

There are two restrictions in using texture maps on the devices.

- The texture map used by the device must have a size which is a power of two in both directions (the s size and t size need not be equal) and not larger than 512. If a texture map is specified with (define_texture) whose axis size is not a power of two and is less than 512, then that axis is "up sampled" to a size which is a power of two. If the non power of two size is larger than 512, then it is "down sampled" to 512.
- The texture map must reside in contiguous off screen memory to be utilized. Every attempt is made to place the user's texture map into off screen memory as it was specified. If sufficient memory cannot be obtained, the texture map is down sampled until sufficient memory can be allocated. This down sampling is flagged by a non-unary return from the `texture_map` or `bf_texture_map` procedure. The value returned is one more than the number of down samplings that were required. If no memory is available, then an error is generated and the default texture map is used.

There are three types of texture mapping on the HP 98736 or HP 98766 devices:

- point sampled (`POINT`)
- rectangular image pyramid (`RIP`) mapping
- environment mapping

These are selectable via the `TEXTURE_CONTROL gescape`. The tradeoffs between the first two types are picture quality and amount of offscreen memory required.

Environment mapping provides a means of mapping a 3-dimensional environment onto a 3-dimensional surface as if that surface was contained in that environment.

Note This new texture mapping mode is discussed in the *Starbase Graphics Techniques Concepts and Tutorials* manual.

The simplest method is to point sample the texture. This is done by computing (s,t) at the center of each pixel. The (s,t) values are then used to find a single point in the texture map. When adjacent pixels take large steps in texture space, aliasing of texture maps becomes a problem.

The aliasing problem is reduced by filtering the texture map. This filtering is accomplished by the rectangular image pyramid method and is paid for by increased use of off screen memory.

An image pyramid map is created by recursively filtering the original texture map along each axis independently. This technique is referred to as a rectangular image pyramid (`RIP`) map. This method has the advantage that the area covered by a pixel in the texture space will be approximated by a rectangle instead of a square. The disadvantage of this approach is that four times the off screen memory is required to store the texture map, this limits the maximum size of your defined texture map to 256x256.

The `TEXTURE_CONTROL gescape` allows you to specify a pre-filtered RIP map so that the application can use its own filtering algorithm. Texture maps can also be explicitly downsampled by the application, by calling the `TEXTURE_DOWNSAMPLE gescape`, and can be retrieved from off screen by the `TEXTURE_RETRIEVE gescape`.

Anti-aliasing

There are two anti-aliasing modes on the devices that may be accessed from the `line_filter` and `perimeter_filter` procedures. The following index values activate the filters in described manner:

`0` One-pixel-wide output. That is, anti-aliasing is turned off.

`1` Three-pixel-wide output. This mode produces three pixels for each major axis increment. A blending value (α) is calculated for each pixel to be drawn and the *pixel_color* is blended with the frame buffer color by: $\alpha \times pixel\_color + (1 - \alpha) \times frame\_buffer\_color$.

`2` Three-pixel-wide output. This mode produces three pixels for each major axis increment. A blending value (α) is calculated for each pixel to be drawn and the pixel_color is blended with the frame buffer color by: $\alpha \times pixel\_color + frame\_buffer\_color$.

Contouring

There are two restrictions to using contouring on the `hp98736` or `hp98766` drivers:

- The only primitives that support contouring are `polyhedron_with_data` and `polygon_with_data3d` in the form of triangular and/or quadrilateral facets.

- The maximum number of entries supported in the contour table is 64. For the *interpolated* contour table case, if more than 64 entries are attempted to be defined, only the first 64 will be recognized and the linear interpolation will be over those first 64 entries. For the *defined* contour table case, if more than 64 entries are attempted to be defined, only the first 64 will be recognized and the scalar contour transitions values for those first 64 entries will be used for contouring.

Since contoured primitives are processed differently from non-contoured primitives, the performance for rendering contoured primitives will differ from non-contoured primitives.

Functional Conflicts

Some functional conflicts exist between the various advanced features supported by the HP 98736 and HP 98766 device drivers. Alpha blending and texture mapping use the same hardware, therefore the two features cannot be enabled simultaneously. When alpha blending is enabled, the device driver will automatically change the `interior_style` from `INT_TEXTURE` to `INT_SOLID`. When texture mapping is enabled, alpha transparency will be disabled by the device driver. Texture mapping and shading can be enabled concurrently, as can alpha transparency and shading. Certain restrictions are necessary for anti-aliased primitives, which are allowed only in `CMAP_FULL` and `CMAP_MONOTONIC` modes. At least eight planes are required for anti-aliasing in `CMAP_MONOTONIC`, and twelve planes for `CMAP_FULL`. Changing to `CMAP_NORMAL` will turn off anti-aliasing, and enabling anti- aliasing in `CMAP_NORMAL` is not allowed. Contouring, deformation, model clipping and capping are all completely independent features which can be combined in any order.

High Performance Bit-Per-Pixel Support

The TurboVRX provides device support for high speed bit-per-pixel block reads and writes. Bit-per-pixel mode is set by using the `R_BIT_MODE gescape`. When in this mode, one byte of data represents data for eight pixels. See the description of this `gescape` for more details.

Multiple-Plane Bit-Per-Pixel Support

The `gescape` `GR2D_PLANE_MASK` defines a mask that allows multiple planes to be read or written. The definition of `GR2D_PLANE_MASK` requires data array space for each plane that will be read or written, and each is done individually. See the `GR2D_PLANE_MASK` definition for more details.

Bit-per-Pixel Replacement Rule per Plane

The TurboVRX supports a replacement rule per plane while doing bit-per-pixel block writes. This allows a replacement rule to be set individually for each plane. See the description of the `gescape` `GR2D_PLANE_RULE` for more details.

Setting up the TurboVRX for the Series 300/400

The `hp98735` and `hp98736` device drivers can be used with the graphics display configured only in DIO-II address space. Refer to the *Graphics Devices* section of the *Installing Peripherals Manual* for a description of DIO-II address space. The graphics interface card may be installed in any DIO II slot in the computer's backplane or in any I/O slot of the expander.

If the TurboVRX is configured as an external display, then there will not be an Internal Terminal Emulator (ITE) for that device. Since it is the ITE that normally initializes the display, external devices must be reset after power-up by running a simple Starbase program with a mode of `RESET_DEVICE` in the `gopen` call. It may also be necessary to run this program after running an application which manipulated the overlay color map, such as a windows application program. An example program which could be called from `/etc/rc` during power-up is given at the end of this section. For more details concerning the effects of `RESET_DEVICE`, see the *Device Initialization* information in this section.

Example Program to Reset the HP 98736

```
/*
 * Starbase program: reset98736.c
 * Compile: cc -o reset98736 reset98736.c -ldd98736 -lsb1 -lsb2 -lm
 * Destination: /usr/bin
 * Execute: add line to the /etc/rc -
 *   /usr/bin/reset98736 /dev/crt.external
 *
 * Example program to be put in /etc/rc for resetting
 * an external HP 98736 device during power-up.
 */
 #include ⟨starbase.c.h⟩

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

    if ((fildes = gopen(argv[1],OUTDEV,hp98736,INIT|RESET_DEVICE))< 0)
       printf("External HP 98736 %s initialization failed.\\n", argv[1]);
    else {
       printf("External HP 98736 %s initialization succeeded.\\n",argv[1]);
       gclose(fildes);
    }
 }
```

DIO-II Switch Settings

The graphics interface card has a single 8-bit address select switch. The switches are labeled (left to right) from MSB to LSB. The leftmost switches represent the most significant bits, hence, switch 1 is the most significant bit of the address, and switch 8 is the least significant. Switches in the open position represent 0's while switches not in the open position represent 1's. See the address table following for the switch-setting to address mapping.

The TurboVRX can only be used in DIO-II address space. In this mode, the eight switches determine the DIO-II select codes to be used. A TurboVRX device will use three DIO-II select codes. Both the frame buffer and control space reside in the select code areas. The control space requires 8 Mbytes of space, starting at `CTL_BASE`. The eight switches described above determine the address of `CTL_BASE`. The frame buffer also requires 8 Mbytes of space, starting at `FB_BASE`.

Table TurboVRX-3. DIO-II Control Space Settings

Switch Setting MSB to LSB	CTL_BASE	FB_BASE	DIO-II Select Code	Device File Minor Number
1000 0100	$0100 0000	$0180 0000	132	$840200
1000 1000	$0200 0000	$0280 0000	136	$880200
1000 1100	$0300 0000	$0380 0000	140	$8c0200
1001 0000	$0400 0000	$0480 0000	144	$900200
1001 0100	$0500 0000	$0580 0000	148	$940200
1001 1000	$0600 0000	$0680 0000	152	$980200
1001 1100	$0700 0000	$0780 0000	156	$9c0200
1010 0000	$0800 0000	$0880 0000	160	$a00200
1010 0100	$0900 0000	$0980 0000	164	$a40200
1010 1000	$0a00 0000	$0a80 0000	168	$a80200
1010 1100	$0b00 0000	$0b80 0000	172	$ac0200
1011 0000	$0c00 0000	$0c80 0000	176	$b00200
1011 0100	$0d00 0000	$0d80 0000	180	$b40200
1011 1000	$0e00 0000	$0e80 0000	184	$b80200
1011 1100	$0f00 0000	$0f80 0000	188	$bc0200
1100 0000	$1000 0000	$1080 0000	192	$c00200
1100 0100	$1100 0000	$1180 0000	196	$c40200
1100 1000	$1200 0000	$1280 0000	200	$c80200
1100 1100	$1300 0000	$1380 0000	204	$cc0200
1101 0000	$1400 0000	$1480 0000	208	$d00200
1101 0100	$1500 0000	$1580 0000	212	$d40200
1101 1000	$1600 0000	$1680 0000	216	$d80200
1101 1100	$1700 0000	$1780 0000	220	$dc0200
1110 0000	$1800 0000	$1880 0000	224	$e00200
1110 0100	$1900 0000	$1980 0000	228	$e40200
1110 1000	$1a00 0000	$1a80 0000	232	$e80200
1110 1100	$1b00 0000	$1b80 0000	236	$ec0200
1111 0000	$1c00 0000	$1c80 0000	240	$f00200
1111 0100	$1d00 0000	$1d80 0000	244	$f40200
1111 1000	$1e00 0000	$1e80 0000	248	$f80200
1111 1100	$1f00 0000	$1f80 0000	252	$fc0200

A DIO-II display may be used as the system console or as an external display. In order to use the display as the system console, it must be configured as the first DIO-II display in the system, and there must be no DIO-I console, or remote terminals. Being the first DIO-II device means that it has the lowest DIO-II select code in the system. In order to use a TurboVRX as a DIO-II system console, select code 132 is recommended.

Bus Master Daisy Chain

The Graphics Interface card for the TurboVRX supports DMA transfers. This capability requires the interface card to be inserted into the bus master daisy chain. The daisy chain position is controlled on the interface card by the set of jumpers closest to the DIO-II connector. The jumper position selects one of the following positions: A, B, C, D, E, and F. The CPU card also has bus master daisy chain jumpers, which can be set to MC, A, B, and C. The card must always be the highest position in the daisy chain, and no intervening unused positions in the chain are allowed. The table below shows the suggested positions for the master bus daisy chain. The CPU DMA controller card resides at position D, so that position should not be used for the Graphics Interface. If your system is utilizing DOS or VME controllers, check the respective documentation for daisy chain position requirements.

Table TurboVRX-4. DIO-II Control Space Settings

Interfaces	Interface 1 Position	Interface 2 Position	Interface 3 Position
2	D	E	———
3	D	E	F

Caution Failure to select the daisy chain positions can cause the system to panic or halt unexpectedly!

System Parameters

It is necessary to increase some of the HP-UX tunable system parameters due to the size of the DIO-II mapping of a TurboVRX. For details on how to reconfigure your kernel, refer to the *HP-UX System Administrators Manual* (particularly the *Configuring HP-UX* section in *The System Administrators Toolbox* and the *System Parameters Appendix*.

It is essential that you consult the above referenced HP-UX documentation before you attempt to reconfigure your system. It is possible to adversely affect your HP-UX system if a mistake is made. Make sure you have an understanding of these procedures before proceeding.

Setting up the TurboVRX on the Series 700

The HP98765A/HP98766A graphics interface card may be installed in any SGC slot in the computer's backplane or in any I/O slot of the expander. There are no user adjustable settings on this interface card.

If the HP 98766 is configured as an external display, then there will not be an Internal Terminal Emulator (ITE) for that device. Since it is the ITE that normally initializes the display, external devices must be reset after power-up by running a simple Starbase program with a mode of `RESET_DEVICE` in the `gopen` call. It may also be necessary to run this program after running an application which manipulated the overlay color map, such as a windows application program. An example program which could be called from `/etc/rc` during power-up is given at the end of this section. For more details concerning the effects of `RESET_DEVICE`, see the *Device Initialization* information in this section.

Example Program to Reset the HP 98766

```
/*
 * Starbase program: reset98766.c
 * Compile: cc -o reset98766 reset98766.c -ldd98766 -lsb1 -lsb2 -lm
 * Destination: /usr/bin
 * Execute: add line to the /etc/rc -
 *   /usr/bin/reset98766 /dev/crt.external
 *
 * Example program to be put in /etc/rc for resetting
 * an external HP 98766 device during power-up.
 */
 #include ⟨starbase.c.h⟩

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

    if ((fildes = gopen(argv[1],OUTDEV,hp98766,INIT|RESET_DEVICE))< 0)
       printf("External HP 98766 %s initialization failed.\\n",argv[1]);
    else {
       printf("External HP 98766 %s initialization succeeded.\\n",argv[1]);
       gclose(fildes);
    }
 }
```

Note A SGC display may be used as the system console or as an external display.

Special Device Files (`mknod`)

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod(1M)` information in the *HP-UX Reference* for further information. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory.

Any name may be used for the special device file, however the name that is suggested for these devices is `crt`. The following examples will create a special device file for this device. Remember that you must be superuser (the root login) to use the `mknod` command.

`mknod` on the Series 300

Since the device is in DIO-II address space, (refer to the Switch settings section) the `mknod` parameters should create a character special device with a major number of `12` and a minor number of `0xSc0200` where `Sc` is the external select code in hexadecimal notation.

```
mknod /dev/crt  c  12  0xSc0200
```

The `hp98736` device driver may also be used for the overlay planes in graphics mode. The minor number may be set to cause Starbase drivers to use either three or four overlay planes. Note that since the terminal emulator and the window system operate in the overlay planes also, there will be interactions with these processes if a graphics driver is opened in this manner while these processes are present.

To create a device file that will allow the HP 98736 device to use only three overlay planes, the following command should be used:

```
mknod /dev/ocrt  c  12  0xSc0201
```

To create a device file that will allow the HP 98736 device to use all four overlay planes, the following command should be used:

```
mknod /dev/o4crt c 12 0xSc0203
```

The `Sc` in all of these example is determined by the switch settings on the host interface, as previously described in the DIO-II Control Space table.

`mknod` on the Series 700

Following are some examples of using the mknod entry for the HP-UX Operating System.

For an SPU with only one SGC interface slot (e.g. Model 720), a sample mknod entry would be:

```
/etc/mknod /dev/crt c 12 0x100000
```

For an SPU with two SGC interface slots, a sample mknod entry for the other slot would be:

```
/etc/mknod /dev/crt c 12 0x000000
```

Linking the Driver

Shared Libraries

The `/usr/lib` directory contains the shared device driver files named: `libdd98735.sl`, `libdd98736.sl`, `libdd98765.sl` and `libdd98766.sl`.

The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Archive Libraries

The archive device driver is located in the `/usr/lib` directory with the file names: `libdd98735.a`, `libdd98736.a` `libdd98765.a`, or `libdd98766.a`.

You can link the device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libdd`<*device driver*> `* an appropriate relative path name * the -ldd`<*device driver*> option with the `LDOPTS` environmental variable exported and set to `-a archive`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

This driver also requires the math library to be linked with C programs. All programs must also be linked with the Starbase graphics libraries `/usr/lib/libsb1.a` and `/usr/lib/libsb2.a`, or use the `-l` option `-lsb1` and `-lsb2` . The device driver needs to precede the graphics libraries when linking, as shown in the examples below.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldd<device driver> -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldd<device driver> -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldd<device driver> -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

The parameters to `gopen` determine which device driver and display device are actually used.

X Windows

These display systems are designed to provide accelerated Starbase graphics within the X Window System environment. The display systems support the X Window System by providing window clipping, multiple color maps, and multiple display modes.

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 10-3. HP 98735 and HP 98736 Display Types

Series 300/400	TurboVRX [T1]	HP 98735A	High-Res Color
Series 300/400	TurboVRX [T2]	HP 98736A	High-Res Color
Series 300/400	TurboVRX [T3]	HP 98736B	High-Res Color

Table 10-4. HP 98765 and HP 98766 Display Types

Series 700	TurboVRX [T2]	HP 98765A	High-Res Color
Series 700	TurboVRX [T4]	HP 98766A	High-Res Color

The supported server modes are Combined and Overlay.

Table 10-5. Windows in Overlay Planes

Contents of X0screens		**Visual Class**	**Backing Store**		**Comments**
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
3	No	PseudoColor	●	●	one color reserved for transparency in combined mode
4	No	PseudoColor	●	●	one color reserved for transparency in combined mode

The supported server modes are Combined and Image.

Table 10-6. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	
16	No				Not supported
	Yes (8/8)	PseudoColor	●		
24	No	DirectColor	●		
	Yes (12/12)	PseudoColor	●		12-bit indexing
		DirectColor	●		

X11 Cursors and Starbase Echos

The following list shows default positions where the Starbase echo and X11 cursor (called echo and cursor, respectively) reside for each of the X11 server operating modes.

TurboVRX Display

- Overlay Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

  If image planes are opened and X11 uses three overlay planes, vector echo resides in cursor plane.

  If image planes are opened and X11 uses four overlay planes, vector echo resides in image planes.

  X11 cursor uses hardware cursor.

- Image Mode

  If image-plane X11 window is opened, raster echo resides in image planes and vector echo resides in cursor plane.

X11 cursor uses hardware cursor.

- Stacked Screen Mode

  Not supported.

- Combined Mode

  If overlay-plane X11 window is opened, echo shares three or four overlay planes.

  If image-plane X11 window is opened, raster echo resides in image planes.

  If image-plane X11 window is opened and X11 uses three overlay planes, vector echo resides in cursor plane.

  If image-plane X11 window is opened and X11 uses four overlay planes, vector echo resides in overlay planes.

  X11 cursor uses hardware cursor.

Usage and Restrictions

The HP 98735 and HP 98765 Device Drivers do *not* support:

- Z-buffering
- Shading
- The transform engine(s)
- Texture mapping
- Contouring
- Deformation
- Anti-aliasing
- Model clipping
- Capping

If the device is opened to the overlay planes, these device drivers also do *not* support:

- Bank switching.
- Double buffering when using three overlay planes. (Double buffering in the overlay planes is supported to four overlay planes).

Transparency Index

There are four overlay planes in the HP 98735 and HP 98765 displays. Even though these planes can display 16 colors simultaneously, only 15 are available because one color is reserved for the transparency color. By default, this color is index 7 or 15, depending on the overlay depth. When the transparency color's index is written into the overlay planes, the observed color is that of the image planes.

X Window System See Through Color

The X Window system always uses color 7 (15 for 4-plane devices) as the see-through color. This cannot be changed.

HP 98736 and HP 98766

When a device file for the overlay planes is used at gopen time, many Starbase features will be unavailable because of the small number of frame buffer planes. These include shading, anti-aliasing, texture mapping, depth cueing, and bank switching.

A maximum of 32 distinct gopen calls may be initiated simultaneously using the `hp98736` or `hp98766` device drivers from any combination of one or more processes. Additional processes may use the unaccelerated device via the device driver, which has no limit on the number of gopens.

Cursors

The `hp98736` and `hp98766` device drivers implement cursors using either the hardware cursor or overlayed software cursors. If no processes have opened all four overlay planes, then the fourth overlay plane is used for overlayed software cursors.

You can think of the fourth overlay plane used for cursors as a separate *cursor plane*. Any data in the cursor plane will be displayed *over* data in the graphics planes. Data in the other three overlay planes will be displayed *over* data in the graphics planes and the cursor plane. For example, suppose a graphics application is running in the graphics planes while the window manager is running in three of the overlay planes. If the application has a Starbase cursor in the overlay cursor plane, then the cursor will always be visible inside regions of see-thru because the cursor has display priority over the graphics. If the cursor is moved outside of regions of see-thru then it is not visible since the non-see-thru regions in the overlay planes have display priority over the cursor plane.

The HP 98736 and HP 98766 Display Stations have a hardware cursor that supports all cursor types except rubber-band line and rubber-band box. There is only one hardware cursor available. Usage of the hardware cursor is defined as follows:

- If an application is running in a Starbase environment only (i.e. X Windows are not running), then the hardware cursor is given to the first process that attempts to use a raster or full screen crosshair cursor.
- By default, if the X Window system is running, then the window system gets usage of the hardware cursor.
- There is a mechanism for the user to control usage of the hardware cursor via the `gescape`, `R_ECHO_CONTROL`. This `gescape` is discussed in the appendix of this manual.

If the hardware cursor is already being used by another process, then overlayed software cursors are used by the `hp98736` or `hp98766` driver. If the fourth overlay plane is not available for cursors, then an error will be generated when any attempts are made to turn on the cursors.

If a process is using the hardware cursor and it switches to using a non-raster cursor, it retains control of the hardware cursor, but the cursor is drawn in the fourth overlay plane using software. If the fourth overlay plane is not available for cursors, an error is generated and non-raster cursors cannot be used. When the process switches back to a raster cursor, it will again use the hardware cursor.

If an application never uses cursors or uses non-raster cursors exclusively, the driver will never attempt to allocate the hardware cursor. However, once the driver has allocated the hardware cursor, the driver does not relinquish control of the hardware cursor until gclose time. While it is not being used, it simply remains inactive, no other process can use the hardware cursor once it has been assigned to a process.

If allocation of the hardware cursor was not successful, then resources for the software cursor area are allocated (that is, offscreen areas for raster echo definitions). The following functions will cause the driver to attempt to allocate either the hardware cursor or software cursor resources:

- `echo_type`
- `define_raster_echo`
- `R_DEF_ECHO_TRANS`
- `R_ECHO_MASK`
- `R_ECHO_FG_BG_COLORS`
- `R_OV_ECHO_COLORS`

Once and application has used a rubber-band echo type, it will thereafter use only software cursors, even if the echo type is switched back to a non-rubber-band type.

Device Initialization

Parameters for `gopen`

- **Path** - This is the name of the special device file created by the `mknod` command as specified in the device setup section (for example, `/dev/crt`.)
- **Kind** - This indicates the I/O characteristics of the device, which may be one of the following types:

`INDEV`	Input only.
`OUTDEV`	Output only
`OUTINDEV`	Input and Output.

  Input mode is only possible when the driver is opened to an X window.
- **Driver** - The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire the device and by default load the accelerated driver. For example:

`NULL`	for C
`char(0)`	for FORTRAN77
''	for Pascal

Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is ignored. For example:

For the Series 300

`"hp98736"`	for C
`'hp98736'//char(0)`	for FORTRAN77
`'hp98736'`	for Pascal

For the Series 700

`"hp98766"`	for C
`'hp98766'//char(0)`	for FORTRAN77
`'hp98766'`	for Pascal

- **Mode** - The mode control word consists of several flag bits ORed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure.
  - `SPOOLED` - Cannot be used on raster devices, therefore this flag has no effect with this driver.
  - `0` - Open the device without clearing the screen. This will set the color map mode to `CMAP_NORMAL`, but will not initialize the color map itself. In an X window, the color map that was associated with the window before `gopen` will be used by Starbase, without initialization.
  - `INIT` - Open and initialize the device as follows:
    1. Clear overlay planes if opened to overlay planes.
    2. Clear image planes if opened to image planes.
    3. Reset the color map to its default values (opening to the image planes does not affect the overlay color map).
    4. Enable the display for reading and writing.
    5. In an X window, a new color map is created and initialized.
    6. If opening the image planes, set the overlay plane background color (pixel value 0) to be transparent.
  - `RESET_DEVICE` - Open and reset the device as follows:

1. Reset the graphics device hardware.
2. Clear overlay planes if opened to overlay planes.
3. Clear image planes if opened to image planes.
4. Make all color map entries for overlay planes transparent if opened to image planes.
5. Reset the color map to its default values.
6. Enable the display for reading and writing.
7. In an X window, a new color map is created and initialized.

Note The `RESET_DEVICE` flag bit should be used with caution, as it may adversely affect any other processes using the device. This flag bit is intended to reset a device completely and should only be necessary for devices in an unknown state, such as a device powered up in an external I/O space. Most programs should not use this flag bit.

Syntax Example

To open and initialize a TurboVRX device for output:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV,NULL,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,'',INIT);
```

Special Device Characteristics

For Device Coordinate operations, location (0,0) is the upper-left corner of the screen or window with x-axis values increasing to the right and y-axis values increasing down. The lower-right corner of the full-screen visible display is therefore (1279,1023).

Offscreen Memory Usage

Offscreen memory is managed by a global resource manager to insure that multiple processes do not step on each other when using the offscreen. Offscreen is used by the device driver for:

- polygon fill patterns
- texture maps
- raster echo definitions
- raster character fonts
- texture maps
- X Window system (This uses offscreen memory extensively)

Refer to the `gescape` manual pages `R_OFFSCREEN_ALLOC` and `R_OFFSCREEN_FREE` for information on using the offscreen areas for application use.

Device Defaults

Number of Color Planes

When the `gopen` procedure is called, this driver asks the device for the number of color planes available. This number can be either 3 or 4 overlay planes, or 24 image planes. The device driver then acts accordingly.

Dither Default

The default number of colors searched for in a dither cell is 2. The number of colors allowed in a dither cell is 1, 2, 4, 8 or 16.

Raster Echo Default

The default raster echo is the 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 allowed for a raster echo is 64×64 pixels. The default drawing mode for the raster echo is 7 (or).

Color Planes Defaults

For a Starbase gopen to the image planes, the default configuration is an 8-plane color mapped system. By default, all planes in the first bank are display enabled as well as write enabled. For a Starbase `gopen` to the overlay planes, the default is a 3-plane or 4-plane system, with the opened planes write and display-enabled.

For an X window, the default is consistent with the Visual type of the window being opened.

Semaphore Default

Semaphore operations are enabled.

Line Type Default

The default line types are created with the bit patterns shown below:

Line Type	Pattern	Hexadecimal Pattern
0	11111111 11111111	$FFFF
1	11111111 00000000	$FF00
2	10101010 10101010	$AAAA
3	11111111 11111010	$FFFA
4	11111111 11101010	$FFEA
5	11111111 11100000	$FFE0
6	11111111 11110110	$FFF6
7	11111111 10110110	$FFB6

Default Color Map

For Starbase `gopen` calls, if the fourth gopen parameter is zero (`0`) then the current hardware color map is used. For X window `gopen` calls, the currently *associated* color map is used.

If the fourth gopen parameter is `INIT`, then the color map is initialized to the default values shown below:

Table HP98736-66-5. 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

Use the `inquire_color_map` procedure to see the rest of the 255 colors.

When `INIT` is used in the `shade_mode` procedure call, the color map initialization is based on the value of the mode parameter:

`CMAP_NORMAL`	Same as the table above. Only one bank can be displayed at a time.

CMAP_MONOTONIC The color map will be initialized as:

```
for (i=0; i<256; i++) {
   cmap[i]red =
   cmap[i]green =
   cmap[i]blue = i/255.0;
}
```

Only one bank can be displayed at a time.

CMAP_FULL The color map will be initialized as 8 bits red, 8 bits green, and 8 bits blue. All three banks of eight planes will be displayed.

Red, Green and Blue

Each file descriptor opened as an output device has a color table associated with it. If multiple file descriptors are open to the same device, the color table and the device's color map may not always be identical. The color table does not track the color map if the device's color map is changed via another file descriptor path.

For Starbase procedures that have parameters for red, green and blue, the way the actual color is chosen depends on the current `shade_mode` setting.

CMAP_NORMAL The color map is searched for the color that is closest in RGB space to the one requested. That color map index is written to the frame buffer for subsequent output primitives. It is more efficient to select a color with an index rather than specifying a color with red, blue, and green values in this mode because it takes extra time to figure out which index in the color table most closely matches the specified color. In the case of a call to `fill_color`, a dither cell will be computed that most closely approximates the requested RGB combination, using the number of color indexes permitted by `fill_dither`.

CMAP_MONOTONIC The red, green, and blue value is converted to an intensity value using the equation:

```
0.30*red+0.59*green+0.11*blue
```

This intensity is converted to an index value by mapping intensity 0.0 to the minimum index set be `shade_range` and intensity 1.0 to the maximum index set by shade_range. This mode is useful for displaying a high quality monochrome picture on an 8-plane system from data that produces a high quality color picture on a 24-plane system.

CMAP_FULL The color is converted to a color map index by the equation:

```
index=(round(red*32767)>>7) & 0xE0 |
(round(green*32767)>>10) & 0x1C |
(round(blue*32767)>>13)
```

This equation will be used in this mode regardless of whether the user has modified the color map.

Note Multiple gopen calls of an X window with the mode parameter set to zero will share the same X color map resource.

Starbase Functionality

Commands not Supported on the HP 98735 and HP 98765

- alpha_transparency
- backface_control
- bf_alpha_transparency
- bf_control
- bf_texture_index
- contour_enable
- define_contour_table
- define_texture
- deformation_mode
- depth_cue
- hidden_surface
- light_ambient
- light_model
- light_source
- light_switch
- line_filter
- perimeter_filter
- set_capping_planes
- set_model_clip_indicator
- set_model_clip_volume
- shade_range
- surface_model
- texture_index
- texture_viewport
- texture_window
- viewpoint
- zbuffer_switch

Exceptions to Standard Starbase Support

HP 98735 and HP 98765

The following commands are supported under the listed conditions:

`block_read`, `block_write`	The raw parameter for the `block_read` and `block_write` commands is used to enable the use of the bit per pixel mode specified via the `R_BIT_MODE gescape`. The raw parameter for the `block_write` command is used to enable the use of the skipcount specified via the `BLOCK_WRITE_SKIPCOUNT gescape`.
`pattern_define`	4×4 is the largest supported pattern.
`shade_mode`	The color map mode may be selected, but shading cannot be turned on. Dithering is available through the `hp98735` or `hp98765` driver for 12-bit indexing mode.
`text_precision`	Only STROKE_TEXT precision is supported.
`vertex_format`	The use parameter must be zero, any extra coordinates supplied will be ignored.

HP 98736 and HP 98766

The following commands are supported under the listed conditions:

`block_read`, `block_write`	The *raw* parameter for these commands is used to enable the use of the bit per pixel mode specified via the `R_BIT_MODE gescape`. The *raw* parameter for `block_write` can also be used to enable use of the skipcount specified via the `BLOCK_WRITE_SKIPCOUNT gescape`. If `bank_switch` is used to select bank 3 (the Z-bank), all subsequent block writes and block reads will be done using word per pixel mode. This means that `block_write` and `block_read` will assume that the data is organized in 32 bit words,

and (w * h) words will be transfered between the Z-buffer and system memory. The buffer address should be point to a word boundary. Words that are read from the the Z-buffer will contain the Z-buffer data in the 3 most significant bytes of each word, with the least significant byte undefined. Similarly, data being written to the Z-buffer should contain the data to be written in the three most significant bytes of each word.

`inquire_fb_configuration` An HP 98736 device running the HP 98736 device driver, or an HP 98766 device running the HP 98766 device driver, will report ⟨*image_banks*⟩ as four, since the system has 24 display planes and 24 Z-buffer planes (the Z-buffer behaves as one bank even though it is 24 planes deep). The dedicated Z-buffer can be accessed with `block_write`, `block_read`, and `block_move`. The Z-buffer may also be selected for read/write using `bank_switch`. The Z-buffer may not be displayed. The graphics accelerator cannot render to the Z-buffer, for example, polygons or other drawing primitives.

`inquire_current_position` When rendering lines of width greater than 0, the current position is not valid. Set `line_width` to zero (0) and perform a `move2d` to ensure the current position is valid.

`interior_style` If the polygon fill type is INT_HATCH then the following functionality will not work correctly:

- Hidden surface removal.
- Shading and lighting.
- Depth cueing.
- Backfacing attributes and culling.
- Texture Mapping

Splines, polyhedra, quadrilateral meshes, and triangular strips will not be hatched. Performance is also degraded in this mode.

`shade_mode`	Dithering is not available in `CMAP_MONOTONIC` 12 bit indexing mode. Attempting to dither with more than one color in this mode will yield the same results as dithering with one color. Dithering is available through the `hp98735` or `hp98765` drivers for 12 bit indexing mode.
`text_precision`	Only STROKE_TEXT precision is supported.

Number of Light Sources

The `hp98736` and `hp98766` device drivers support up to fifteen point light sources, plus one ambient light source.

Fast Alpha and Font Manager

The `hp98735` and `hp98765` device drivers support raster text calls from the fast alpha and font manager libraries. These calls may be made while running in the overlay or image planes. Since raster fonts consist of one byte per pixel, image plane raster text is written only to the currently selected bank. This is similar to the operation of other raster functions, such as `block_write`. Fast alpha and font manager fonts can be optimized. See the *Fast Alpha/Font Manager's Programmer's Manual* for further information.

Th HP 98736 and HP 98766 device drivers do *not* support raster text calls from the Fast Alpha and Font Manager library.

Gescapes

The following `gescape` functions are supported by the TurboVRX device drivers:`hp98735`, `hp98736`, `hp98765` and `hp98766`. Detailed information about these functions can be found in the appendix of this manual.

- `BLINK_INDEX`—Blink an individual colormap index
- `BLINK_PLANES`—Blink selected image planes (Blink rate is 3.75 Hz)
- `BLOCK_WRITE_SKIPCOUNT` - Specify byte skip count during block write
- `GR2D_DEF_MASK`—Define mask for three operand replacement rule
- `GR2D_MASK_ENABLE`—Enables three operand replacement rule and current mask
- `GR2D_MASK_RULE`—Define three operand replacement rule
- `GR2D_OVERLAY_TRANSPARENT`—Turns on/off transparency of zero valued pixels
- `GR2D_PLANE_MASK`—Enable multi-plane bit/pixel block read/writes
- `GR2D_PLANE_RULE`—Define an independent replacement rule per plane
- `GR2D_REPLICATE`—Allows pixel replication during block moves
- `INQ_12_BIT_INDEXING`—Indicates if display mode is 12 bit indexing
- `PAN`—Places display in 1024×1024 mode and shifts upper left corner 1024 pixels to the right in frame buffer.
- `PLUG_ACCELERATED_PIPELINE`—Controls the rendering of the graphics accelerators into the frame buffer.
- `R_BIT_MASK`—Defines a bit mask for bit per pixel operations
- `R_BIT_MODE`—Enables and disables bit per pixel mode
- `R_DEF_ECHO_TRANS`—Define raster echo transparency
- `R_DEF_FILL_PAT`—Define fill pattern
- `R_ECHO_CONTROL`—Control hardware cursor allocation
- `R_ECHO_FG_BG_COLORS`—Define cursor attributes
- `R_ECHO_MASK`—Define cursor mask
- `R_FULL_FRAME_BUFFER`—Allow access to full frame buffer
- `R_GET_FRAME_BUFFER`—Read frame buffer address
- `R_GET_WINDOW_INFO`—Return frame buffer address of X window
- `R_LINE_TYPE`—Define line style and repeat length
- `R_LOCK_DEVICE`—Lock device
- `R_OFFSCREEN_ALLOC`—Allocates offscreen frame buffer memory
- `R_OFFSCREEN_FREE`—Frees allocated offscreen frame buffer memory
- `R_OV_ECHO_COLORS`—Select overlay echo colors
- `R_TRANSPARENCY_INDEX`—Specify transparency index
- `R_UNLOCK_DEVICE`—Unlock device
- `READ_COLOR_MAP`—Read color map

- STEREO—Supports stereoscopic display systems.
- SWITCH_SEMAPHORE—Semaphore control

The following gescape functions are unique to the HP 98736 and HP 98766 drivers. Detailed information on these gescapes can be found in Appendix A of this manual.

- CONTOUR_CONTROL—Selects interpretation of scalar data for contouring.
- DC_COMPATIBILITY_MODE—Control compatibility mode for device coordinate primitives.
- DRAW_POINTS—Select different modes of rounding for rendered points.
- ILLUMINATION_ENABLE—Enable or disable illumination data.
- TEXTURE_CONTROL—Select texture map filter type.
- TEXTURE_DOWNSAMPLE—Controls texture map downsampling.
- TEXTURE_RETRIEVE—Read back offscreen filtered texture maps.
- TOGGLE_2D_COLORMAP—Enable or disable 2D colormap mode.

The following gescape functions are supported by the HP 98736 and HP 98766 drivers and work on other accelerated drivers as well. Detailed information on these gescapes can be found in Appendix A of this manual.

- AUTO_RESIZE_DEVICE—Automatically scale graphics output when the window size changes.
- BLOCK_WRITE_SKIPCOUNT—Specify skip count for block writes.
- GAMMA_CORRECTION—Enable/disable gamma correction
- LS_OVERFLOW_CONTROL—Sets options for lighting overflow situations
- PATTERN_FILL—Fill polygon with stored pattern.
- POLYGON_TRANSPARENCY—Define front and backfacing polygon transparency patterns
- RESIZE_DEVICE—Scale graphics output for the current window size.
- TRANSPARENCY—Allows “screen door” for transparency pattern
- Z_WRITE_ENABLE—Allows creation of 3D cursors in overlay

The following gescape functions were available on the `hp98731` device driver, but will *not be supported* on the either the `hp98736` nor `hp98766` device drivers:

- `IMAGE_BLEND`—Enable/disable video blending.
- `OVERLAY_BLEND` —Control blending of overlay plane frame buffer.
- `PAN_AND_ZOOM`—Pixel pan and zoom.
- `R_DMA_MODE`—Changes the definition of the `raw` flag for block writes.
- `SET_BANK_CMAP`—Instell frame buffer bank colormaps.

Performance Tips

General

1. These drivers are tuned for maximum performance on polyline and polygon primitives. If these primitives are sent through the Starbase procedural interface, a performance penalty occurs because of subroutine overhead. For this reason, user macros have been provided to allow the application programmer to obtain full device performance. Fast macros are supported on the HP 98736 and HP 98766 devices for `move2d, draw2d, move3d, draw3d, polyline2d, polyline3d, polygon2d, and polygon3d`. These macros are included in the standard Starbase header file `/usr/include/starbase.c.h`. To enable use of the fast macros, insert the following line above the include statement for the Starbase header file:

   ```
   #define _HP_FAST_MACROS 1
   ```

   Detailed instructions concerning use of the fast macros is presented in the *Output Primitives* section of the *Starbase Graphics Techniques Concepts and Tutorials* manual. The use of macros in encouraged, since performance is significantly better for all primitives that support fast macros. Including fast macros into your application program will significantly increase code size, so it is desirable to minimize primitive calls when fast macros are enabled.

2. As with any driver, buffering is done to enhance performance. This means that graphics commands and primitives are batched before being sent to the device. The `buffer_mode` call defaults to buffering enabled, which provides the best performance. Calling `buffer_mode` to disable buffering will significantly degrade performance by turning off batching. An excessive number of `make_picture_current` or `flush_buffer` calls will also reduce performance substantially. The `make_picture_current` call is especially expensive since the driver will wait until the device has drawn all graphics primitives and interpreted all graphics commands. The `flush_buffer` is much faster because it simply sends all pending commands and data to the device, then returns immediately.

3. Performance optimizations have been made so that sequential calls of the same output primitive with no intervening attribute changes or different primitive calls goes faster. For example, the sequence `polygon, polygon, polyline, polyline` is faster than `polygon, polyline, polygon, polyline`. Also

`line_color, polyline, polyline` is faster than `line_color, polyline, line_color, polyline`. So grouping by primitive and subgrouping primitives by attribute can give substantial performance improvements.

4. Typically, the rendering engine of either the HP 98736 or HP 98766 renders primitives from its internal buffer as the system CPU is doing other things. Substantial performance benefits can be realized from this parallel processing.

   However, certain operations will cause the CPU to wait for the HP 98736 or HP 98766 to finish emptying its buffer. An example of this wait is the `make_picture_current` operation. Also, any operation that inquires information from the HP 98736 may cause this wait to occur. Examples of inquiries are picking and extent testing.
5. For programs which use Z-buffer hidden surface removal with the dedicated Z-buffer, it is much faster to clear the Z-buffer simultaneously with screen clears than to do the clears sequentially. This is accomplished by calling `clear_control` with `CLEAR_ZBUFFER` flag set in the mode word. This ensures that subsequent calls to `clear_view_surface` and `dbuffer_switch` will also clear the Z-buffer. See the manual page for `clear_control` for more details.
6. Performance for 12-bit indexing is poorer if dithering is enabled since a much larger color map is searched to obtain the dither colors than on previous devices. The best performance for 12 bit indexing is obtained by not dithering and by specifying colors by index rather than their R,G,B values.

For the HP 98735 and HP 98765

1. If Starbase echos are overlayed (i.e. in the fourth overlay plane), or hardware cursors are used, then graphics performance is is significantly better since it is not necessary to pick up the cursor each time the frame buffer is updated.
2. Polygons are filled faster when the drawing mode is `SOURCE, NOT_SOURCE,` or `ONE`.
3. Horizontal and vertical lines are faster than diagonal lines on this device since the hardware block mover is used to generate pixels.
4. The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.

5. The performance of `block_read` and `block_write` is significantly better if the width in bytes plus any skipcount specified via `BLOCK_WRITE_SKIPCOUNT` is a multiple of 4.

6. The `hp98735` and `hp98736` drivers automatically use VDMA capabilities for `block_read` and `block_write`, providing up to a 6x improvement in performance. VDMA is used only for byte per pixel `block_read` and `block_write` if the width in bytes plus any skipcount specified via `BLOCK_WRITE_SKIPCOUNT` is a multiple of 4, and the data to be transferred is at least 1024 bytes. *VDMA is not supported on the Series 700.*

Rendering (HP 98736 and HP 98766)

1. When doing shaded polygons, the fewer the features, the faster the polygon generation. Positional viewpoint and light sources can degrade performance.
2. The HP 98736 or HP 98766 rendering engine runs at full speed when rendering flat shaded polygons. Other techniques slow the rendering of polygons on either the HP 98736 or HP 98766. The major features which reduce performance are clipping, depth cueing, hidden surface removal, Gouraud shading, texture mapping, and anti-aliasing. The performance degradation is especially noticeable on large polygons and vectors. Turning on any one of these features can noticeably lower the rendering performance.
3. Using the pattern `gescape` or replacement rules that require extra reads of the frame buffer (for example, `source OR destination`) may also degrade performance.
4. Rendering mode commands such as `hidden_surface`, `shade_mode`, and `double_buffer` can be slow. These commands should not be unnecessarily called. For example, it is not necessary to repeatedly call `hidden_surface` from an animation loop. It is intended that these routines be called to initialize a rendering mode and called again only when the mode needs to change.
5. Due to the subpixel resolution on both the HP 98736 and HP 98766, attempting to render points by moving and drawing from a point to itself will not appear unless the point is rendered on the pixel center. The `gescape` opcode `DRAW_POINTS` has been provided to center such primitives so that points are rendered on pixel centers. Alternatively, the environment variable `HP98736_POINT_COMPATIBILITY_MODE` or `HP98766_POINT_COMPATIBILITY_MODE` can be set to center rendered points so that the application need not be recoded using the `DRAW_POINTS` `gescape`.

Raster Operations (HP 98736 and HP 98766)

- The procedure `block_move` is faster than `block_read` or `block_write` since the hardware frame buffer block mover can be used.
- The performance of `block_read` and `block_write` is significantly better if the width in bytes plus any skipcount specified via `BLOCK_WRITE_SKIPCOUNT` is a multiple of 4.
- The `hp98736` driver automatically uses the VDMA capabilities for `block_read` and `block_write`, providing up to a 6x improvement in performance. DMA is used only for byte per pixel `block_read` and `block_write` if the width in bytes plus any skipcount specified via `BLOCK_WRITE_SKIPCOUNT` is a multiple of 4, and the data to be transferred is at least 1024 bytes. *The "hp98766" driver does not utilize a VDMA mode, but achieves superior performance via the Series 700's SGC bus.*
- If `bank_switch` is used to select bank 3 (the Z bank), all subsequent block writes and block reads will be done using word per pixel mode. This means that `block_write` and `block_read` will assume that the data is organized in 32-bit words, and (w * h) words will be transferred between the Z buffer and system memory. The buffer address should be pointed to a word boundary. Words that are read from the Z buffer will contain the data in the three most significant bytes of each word, with the least significant byte undefined. Similarly, data being written to the Z buffer should be contained in the three most significant bytes of each word.

Cautions

- As mentioned previously, accessing the off-screen portion of the frame buffer (using `gescape` calls) should be done with care, since other processes access this region.
- Certain `gescape` calls should be used with caution since they bypass protection mechanisms used to prevent multiple processes from interfering with each other. For example, since the device resources can only be rationally used by one graphics process at a time, the driver activates a semaphore and locks the device before doing any output. This ensures, for example, that one process will not change the replacement rule while another process is in the middle of filling a polygon. It also prevents the terminal driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when done processing output. Some of the `gescape` listed in this chapter allow the user to change this locking mechanism and should be used with great caution.

The following cautions are provided in using the HP 98736 or HP 98766 accelerated drivers:

- Vertex color or intensity values should range between 0 to 32,767 when used in calls using DC values (e.g. `dcpolygon`).
- The overlay off-screen contains the ITE font (which is regenerated when control-shift-reset is done on the ITE keyboard) and may contain any number of window systems fonts depending on the current window usage.
- Polygons of up to 255 vertices (after clipping) are supported. If a polygon has more than 255 vertices, only the first 255 vertices are displayed.
- This driver's implementation of the `set_capping_planes` procedure requires exclusive access to the accelerated device during a capping sequence. It will lock out any other accelerated file descriptor from accessing the device during such a sequence. Programs making use of the `set_capping_planes` procedure with this driver should limit their Starbase (and X) activities during a capping sequence to Starbase calls to the file descriptor currently capping. These call should be limited to graphics primitives (e.g., `polygon3d`, `polyline3d`, etc.), and rendering attribute changes (e.g., `fill_color`, `perimeter_color`, etc.).
- When using either the HP 98736 device or HP 98766 device with a graphics accelerator it is possible for illegal operations to cause the transform engine or

scan converter hardware to enter an unknown state. If this happens, Starbase will report an error the next time it tries to use the hardware. The user will see this as a Transform engine timed out or Hardware/scan_converter time out error. These are Starbase errors 14 and 52 respectively. If the `hp98736` device driver is being used, then this is a fatal error. When this error is discovered, Starbase reports the error and aborts execution.

- If an application needs to take some emergency action before an untimely termination, such as saving valuable data, then the application should check for these error conditions and take appropriate measures. Errors may be caught by an application using the `gerr_control` procedure described in the Starbase Reference manual.
- It is also possible to avoid the termination completely if the application's error handler does not return control to Starbase. However, it may be impossible to proceed with any graphics efforts using the accelerator until it is reset. Information regarding the state of other `gopen` calls to the accelerated device (for example, the current fill color) may be lost, and other `gopen` calls may abort as well.
- When defining texture maps inside a display list segment, it is critical that the desired shade mode be established outside of the segment. The shade mode is used at texture map definition time to determine the depth of the map. Changing the shade mode inside a display list segment will have no effect on the depth determination of a texture map defined within that segment.

FINAL TRIM SIZE : 7.5 in x 9.0 in

11

The HP 300H Device Driver

Device Description

This device driver is used with Hewlett-Packard Series 300 high-resolution display systems. See the table in the introduction for supported configurations.

The video board for each of these display systems fits in a SPU system slot. These display systems have a resolution of 1024×768 pixels. The monochrome display system has a single plane of frame buffer. The HP 98545A Color Display System has four planes of frame buffer to provide 16 simultaneous colors. The HP 98547A, HP 98549A, and HP 319C+ Color Display Systems have six planes for 64 colors. A color map provides 8 bits per color (for red, green and blue), providing a color palette of over 16 million colors.

These display systems are bit-mapped devices with special hardware for:

- Write enabling planes.
- Displaying planes.
- Writing pixels to the frame buffer with a given replacement rule (see `drawing_mode`).
- Blinking planes.
- Moving a block of pixels from one place in the frame buffer to another.

The monochrome and color displays are organized as an array of bytes, with each byte representing a pixel on the display. For the monochrome display, the Least Significant Bit (LSB) of each byte controls the display with 0 for black (pixel off) and 1 for white (pixel on).

For the color displays, the 6 (4 for HP 98545A) LSBs of each byte determine the color, providing color values from 0-63 (0-15 for HP 98545A). These values are used to address the color map. The color map is a RAM table that has 64 (16 for HP 98545A) addressable locations and is 24 bits wide (8 bits each for red, green and blue). Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

Typically, you do not need to read or write pixels directly into the frame buffer. However, for those applications which require direct access, Starbase provides the `gescape` function `R_GET_FRAME_BUFFER` which returns the virtual memory address of the beginning of the frame buffer. This `gescape` is discussed in the appendix. Frame buffer locations are then addressed relative to the returned address. The first byte of the frame buffer (byte 0) represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1023 is the last (right-most) pixel on the top line. Byte 1024 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 786,431 ($767 \times 1024 + 1023$).

Offscreen Memory

The frame buffer is 1024×1024 bytes. The last 256 lines of the frame buffer are not displayed and are referred to as offscreen memory. Offscreen memory may be accessed via the `gescape` function `R_FULL_FRAME_BUFFER` documented in the appendix. Care should be taken when using this `gescape` since other processes, Starbase and window systems, access the frame buffer offscreen memory.

The HP 300H Device Driver allocates a portion of offscreen memory for fill patterns and echo storage. In a raw environment, the first 16 lines are reserved for Starbase fill patterns and each raster echo will use a 64×192 pixel rectangle. In general, the remaining portions of offscreen are allocated from top to bottom.

X11 uses offscreen for its sprite, fonts, pixmaps and window backing store (retained rasters). In general, X11 uses offscreen memory intensively; therefore, usage of offscreen memory while running X11 is not recommended.

Refer to the *Starbase Graphics Techniques* for information on how this device driver can be used with X11.

HP 98549A and HP 319C+

The HP 98549A and HP 319C+ display may also be accessed using the HP 98550 driver. It has higher performance than the HP 300H driver. Applications using the HP 300H driver should not be run simultaneously with applications using the HP 98550 driver on the same display nor should the HP 300H driver be used in an X11 window. The drivers manage offscreen frame buffer memory differently and will interfere with each other. This also applies to the X11 server which will (by default) use the HP 98550 driver. The X11 server cannot be told to use the

HP 300H driver. Note also that the HP 98549A and HP 319C+ display is only supported by the HP 300H driver when it is configured as the internal display.

Setting Up the Device

Switch Settings

There are no switches to set on the video boards for these devices. However, when these video boards are used with the HP 310 Processor Board, the display disable switch on the processor board must be set. Look at the four switch group near the back plate. If the third switch from the back plate is set such that the dot closest to the display board's edge is down, the internal display is disabled. Refer to the *Upgrade Video Output Board* Installation Note (HP Part Number 98547-90600) for more details.

Special Device Files (mknod)

The `mknod` command creates a special device file which is used to communicate between the computer and the peripheral device. See the `mknod`(1M) information in the *HP-UX Reference* for further details. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

The `mknod` parameters are character device with a major number of 12 and a minor number of 0. Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however the name that is suggested for these devices is `crt`. The following example will create a special device file for this device. Remember that you must be superuser or root to use the `mknod` command. Note that the leading `0x` causes the number to be interpreted hexadecimally.

```
mknod /dev/crt  c  12  0x000000
```

Linking the Driver

Shared Libraries

The shared HP 300H Device Driver is the file named `libdd300h.sl` in the `/usr/lib` directory. The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Archive Libraries

The archive HP 300H Device Driver is located in the `/usr/lib` directory with the file name `libdd300h.a`.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libdd300h.a`
2. an appropriate relative path name
3. the `-ldd300h` option with the `LDOPTS` environmental variable exported and set to `-a archive`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldd300h -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldd300h -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldd300h -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Initialization

Parameters for gopen

The `gopen` procedure has four parameters: Path, Kind, Driver and Mode.

Path	The name of the special device file created by the `mknod` command as specified in the last section, e.g. `/dev/crt`.
Kind	Indicates the I/O characteristics of the device. This parameter must be `OUTDEV` for this driver.
Driver	The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire

the device and by default load the accelerated driver (if applicable). For example:

`NULL`	for C
`char(0)`	for FORTRAN77
''	for Pascal

Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is ignored. For example:

`"hp300h"`	*for C.*
`'hp300h'//char(0)`	*for FORTRAN77.*
`'hp300h'`	*for Pascal.*

Mode — The mode control word consisting of several flag bits which are *or* ed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure.

- `SPOOLED`—Cannot spool raster devices.
- `0` (zero)—Open the device, but do nothing else. The software color map is initialized on monochrome monitors.
- `INIT`—Open and initialize the device as follows:
  1. 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.

Syntax Examples

To open and initialize an HP 300H device for output:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV, NULL,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,'',INIT);
```

Special Device Characteristics

For device coordinate operations, location $(0,0)$ is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is $(1023,767)$.

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 11-1. Display Types

Series 300 MR Mono	HP 98544A/B	Medium-Res Monochrome
Model 318M		
Model 340M		

The supported server mode is Image.

Table 11-2. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
1	No	StaticGray	●	●	

Table 11-3. Display Types

Series 300 MR Color	HP 98545A	Medium-Res Color

The supported server mode is Image.

Table 11-4. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
4	No	PseudoColor	●	●	
	Yes (2/2)	PseudoColor	●	●	

Table 11-5. Display Types

Series 300 MR Color	6-plane Color	HP 98547A	Medium-Res Color

The supported server mode is Image.

Table 11-6. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
6	No	PseudoColor	●	●	
	Yes (3/3)	PseudoColor	●	●	

Starbase Functionality

Commands Not Supported

The following commands are not supported. If one of these commands is used by mistake, it will not cause an error.

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

Commands Conditionally Supported

The following commands are supported under the listed conditions:

block_read, block_write	Note: When using raw mode, be careful not to do a block_read or block_write outside the device limits. When using raw mode without using the R_BIT_MODE gescape, no clipping is performed. See the R_BIT_MODE gescape in the appendix of this manual for more information.

`define_color_table`	On black and white devices, this command defines a software color map, since there is no hardware color map.
`inquire_color_table`	On black and white devices, this command returns the software color map values.
`interior_style`	`INT_OUTLINE` and `INT_POINT` not supported.
`shade_mode`	The color map mode may be selected but shading can not be turned on.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`vertex_format`	The `use` parameter must be zero; any extra coordinates supplied will be ignored.

`with_data`	`partial_polygon_with_data3d`
	`polygon_with_data3d`
	`polyhedron_with_data`
	`polyline_with_data3d`
	`polymarker_with_data3d`
	`quadrilateral_mesh_with_data`
	`triangle_strip_with-data`
	Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

Fast Alpha and Font Manager Functionality

This device driver supports raster text calls from the fast alpha and font manager libraries. See the *Fast Alpha/Font Manager Programmer's Manual* for further information.

Parameters for gescape

The `hp300h` driver supports the following `gescapes`. Detailed information about these functions can be found in Appendix A.

- `SWITCH_SEMAPHORE`—Semaphore control.
- `READ_COLOR_MAP`—Read color map.
- `BLINK_PLANES`—Blink display (blink rate is 2.4 Hz for this device). Not supported in an X11 window.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_GET_WINDOW_INFO`—Returns frame buffer address of Windows/9000 window.
- `R_FULL_FRAME_BUFFER`—Full frame buffer.
- `R_LOCK_DEVICE`—Lock device.
- `R_UNLOCK_DEVICE`—Unlock device.
- `R_BIT_MODE`—Bit mode.
- `R_BIT_MASK`—Bit mask.
- `R_DEF_FILL_PAT`—Define fill pattern.

Performance Tips

Drawing horizontal and vertical lines is faster than drawing diagonal lines on these devices since the hardware block-mover generates the pixels. The procedure `block_move` is faster then `block_read` or `block_write` since the hardware frame buffer block mover can be used.

Cautions

The following cautions are provided in using this driver:

1. As mentioned previously, accessing the off-screen portion of the frame buffer (using `gescape`) should be done with care, since other processes access this region.
2. Certain `gescape` functions should be used with caution since they bypass protection mechanisms used to prevent multiple processes from interfering with each other. For example, since the hardware resources can only be

rationally used by one graphics process at a time, the driver sets a semaphore and locks the device before doing any output. This ensures, for example, that process A will not change the replacement rule while process B is in the middle of filling a polygon. It also prevents the terminal (`tty`) driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when done processing output. Some of the `gescape` functions listed in this chapter allow you to change this locking mechanism and should be used with *great caution.*

The HP 300L Device Driver

Device Description

This device driver is used with Hewlett-Packard Series 300 medium-resolution display systems. See the table in the introduction for supported configurations.

The HP 310 processor board has a built-in medium-resolution monochrome display system. The HP 98542A and HP 98543A video boards fit in an SPU system slot. The monochrome display systems have a single plane of frame buffer. The color display system has four planes of frame buffer to provide 16 simultaneous colors. A color map provides eight bits per color (for red, green and blue), providing a color palette of over 16 million colors.

All three systems have a resolution of 1024×400 pixels; however, this driver treats the display systems as having a resolution of 512×400 pixels since each pixel in the frame buffer has an aspect ratio of 2 to 1. By writing two pixels for every dot to be displayed, square pixels are produced. Some applications or subsystems may use the higher resolution. The `gescape` function `TC_HALF_PIXEL`, documented later in this section, can be used to allow `block_read` and `block_write` access to the full resolution.

These display systems are bit-mapped devices with special hardware for:

- Write enabling planes.
- Displaying planes.
- Writing pixels to the frame buffer with a given replacement rule (see `drawing_mode`).
- Blinking planes.
- Moving a block of pixels from one place in the frame buffer to another.

Both the monochrome and color displays are organized as an array of bytes, with each byte representing a pixel on the display. For the monochrome display, the Least Significant Bit (LSB) of each byte controls the display, with 0 for black (pixel off) and 1 for white (pixel on).

For the color display, the four LSBs of each byte determine the color, providing color values from 0-15. These values are used to address the color map. The color map is basically a RAM table that has 16 addressable locations and is 24 bits wide (eight bits each for red, green and blue). Thus, the pixel value in the frame buffer addresses the color map, generating the color programmed at that location.

Typically, the user does not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase does provide the `gescape` function `R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer. This `gescape` is discussed in the appendix. Frame buffer locations are then addressed relative to the returned address. The first byte of the frame buffer (byte 0) represents the upper left corner pixel of the screen. Byte 1 is immediately to its right. Byte 1023 is the last (right-most) pixel on the top line. Byte 1024 is the first (left-most) pixel on the second line from the top. The last (lower right corner) pixel on the screen is therefore byte number 409599 ($399 \times 1024 + 1023$).

Offscreen Memory

The frame buffer is 1024×512 bytes. The last 112 lines of the frame buffer are not displayed and are referred to as offscreen memory. Offscreen memory may be accessed via the `gescape` function `R_FULL_FRAME_BUFFER` documented in the appendix. Care should be taken when using this `gescape` since other processes, Starbase and window systems, access the frame buffer offscreen memory.

The HP 300L Device Driver allocates a portion of offscreen memory for fill patterns and echo storage. In a raw environment, the first 16 lines are reserved for Starbase fill patterns, and each raster echo will use a 64×384 byte rectangle (64×192 square pixels). The last 16 lines are reserved for Starbase fill patterns. In general, the remaining portions of offscreen are allocated from top to bottom. Fast Alpha and Font Manager also allocate offscreen memory for font storage.

X11 uses offscreen for its sprite, fonts, pixmaps and window backing store (retained rasters). In general, X11 uses offscreen memory very intensively; therefore, usage of offscreen memory while running X11 is not recommended.

Setting Up the Device

Switch Settings

There are no switches to set on the video boards for these devices. However, when the HP 98542A or HP 98543A video boards are used with the HP 310 processor board, the display disable switch on the processor must be set. Look at the four switch group near the back plate. If the third switch from the back plate is set such that the dot closest to the display board's edge is down, the internal display is disabled. Refer to the *Upgrade Video Output Board* Installation Note (HP Part Number 5958-4342) for more details.

Special Device Files (mknod)

The `mknod` command (see `mknod(1M)` man page), creates a special device file which is used to communicate between the computer and the peripheral device. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

The `mknod` parameters are character device with a major number of 12 and a minor number of 0. Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file; however, the name that is suggested for these devices is `crt`. The following example will create a special device file for this device. Remember that you must be superuser or root to use the `mknod` command. Note that the leading `0x` causes the number to be interpreted hexadecimally.

```
mknod /dev/crt c 12 0x000000
```

Linking the Driver

Shared Libraries

The shared HP 300L Device Driver is the file named `libdd300l.sl` in the `/usr/lib` directory. The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Archive Libraries

The HP 300L Device Driver is located in the `/usr/lib` directory with the file name `libdd300l.a`. This device driver may be linked to a program using the absolute path name `/usr/lib/libdd300l.a`, an appropriate relative path name, or the −`l` option −`ldd300l` with the `LDOPTS` environmental variable set to `-a archive`.

The reason for using the `LDOPTS` environmental variable is that the `-l` option will look for a shared library driver first and then look for the archive driver if shared was not found. By exporting the `LDOPTS` variable as specified above, the `-l` option will only look for archive drivers. For more information, refer to the *Programming on HP-UX* manual on linking shared or archive libraries.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldd300l -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldd300l -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldd300l -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Initialization

Parameters for gopen

The `gopen` procedure has four parameters: Path, Kind, Driver and Mode.

Path	The name of the special device file created by the `mknod` command as specified in the last section, e.g., `/dev/crt`.
Kind	Indicates the I/O characteristics of the device. This parameter must be `OUTDEV` for this driver.

Driver
: The character representation of the driver type. This parameter may be `NULL` for linking shared or archive libraries - `gopen` will inquire the device and by default load the accelerated driver (if applicable). For example:

`NULL`	for C
`char(0)`	for FORTRAN77
"	for Pascal

Alternatively, a character string may be used to specify a driver. In this case the `UNACCELERATED/ACCELERATED` flag is overidden. For example:

`"hp300l"`	*for C.*
`'hp300l'//char(0)`	*for FORTRAN77.*
`'hp300l'`	*for Pascal.*

Mode
: The mode control word consisting of several flag bits *or* ed together. Listed below are those flag bits which have device-dependent actions. Those flags not discussed below operate as defined by the `gopen` procedure.

- `SPOOLED`—cannot spool raster devices.
- `0`—open the device, but do nothing else. The software color map is initialized on monochrome monitors.
- `INIT`—open and initialize the device as follows:
  1. Frame buffer is cleared to 0s.

2. The color map is reset to its default values.
3. The display is enabled for reading and writing.

Syntax Examples

To open and initialize an HP 300L device for output:

For C Programs:

```
fildes = gopen("/dev/crt",OUTDEV,INIT);
```

For FORTRAN77 Programs:

```
fildes = gopen('/dev/crt'//char(0), OUTDEV,char(0),INIT)
```

For Pascal Programs:

```
fildes = gopen('/dev/crt',OUTDEV,'',INIT);
```

Special Device Characteristics

For device coordinate operations, location (0, 0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is (511, 399).

X Windows

Supported X Windows Visuals

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 the *Starbase Graphics Techniques* manual.

12

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth=" specification as the first entry in that row of the table.

Table 12-1. Display Types

Series 300 LR Mono	HP 98542A	Low-Res Monochrome
Model 310	Integrated Graphics	Low-Res Monochrome

The supported server mode is Image.

Table 12-2. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
1	No	StaticGray	●	●	
	Yes (3/3)	PseudoColor	●	●	

Table 12-3. Display Types

Series 300 LR Color	HP 98543A	Low-Res Color

The supported server mode is Image.

Table 12-4. Windows in Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
4	No	PseudoColor	●	●	
	Yes (2/2)	PseudoColor	●	●	

Starbase Functionality

Commands Not Supported

The following commands are not supported. If one of these commands is used by mistake, it will not cause an error.

```
backface_control              depth_cue_range
bank_switch                   hidden_surface
bf_control                    interior_style (INT_OUTLINE)
bf_fill_color                 interior_style (INT_POINT)
bf_interior_style             light_ambient
bf_perimeter_color            light_attenuation
bf_perimeter_repeat_length    light_model
bf_perimeter_type             light_source
bf_surface_coefficients       light_switch
bf_surface_model              shade_range
define_trimming_curve         surface_model
depth_cue                     surface_coefficients
depth_cue_color               viewpoint
                              zbuffer_switch
```

Commands Conditionally Supported

The following commands are supported under the listed conditions:

`block_read`, `block_write`	Note: When using `raw` mode, be careful not to do a `block_read` or `block_write` outside the device's limits. When using `raw` mode without the `R_BIT_MODE gescape`, no clipping is performed. See the `R_BIT_MODE gescape` in the appendix of this manual for more information.
`define_color_table`	Since there is no hardware color map on black and white devices, this command defines a software color map.

`inquire_color_table`	On black and white devices, this command returns the software color map values.
`text_precision`	Only `STROKE_TEXT` precision is supported.
`shade_mode`	The color map mode may be selected, but shading can not be turned on.
`vertex_format`	The `use` parameter must be zero; any extra coordinates supplied will be ignored.

Fast Alpha and Font Manager Functionality

This device driver supports raster text calls from the fast alpha and font manager libraries. See the *Fast Alpha/Font Manager Programmer's Manual* for further information.

Parameters for gescape

The `hp300l` driver supports the following `gescapes`. Detailed information about these functions can be found in the Appendix A.

- `SWITCH_SEMAPHORE`—Semaphore control.
- `READ_COLOR_MAP`—Read color map.
- `BLINK_PLANES`—Blink display (blink rate is 2.4 Hz for this device.) Not supported in an X11 window.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_GET_WINDOW_INFO`—Returns frame buffer address of Window/9000 window.
- `R_FULL_FRAME_BUFFER`—Full frame buffer.
- `R_LOCK_DEVICE`—Lock device
- `R_UNLOCK_DEVICE`—Unlock device.
- `R_BIT_MODE`—Bit mode.
- `R_BIT_MASK`—Bit mask.
- `R_DEF_FILL_PAT`—Define fill pattern.

The `gescape` function `TC_HALF_PIXEL` is unique to this driver. Detailed information can be found in the appendix.

Performance Tips

Horizontal and vertical lines are faster then diagonal lines on these devices since the hardware block mover is used to generate the pixels. The procedure `block_move` is faster then `block_read` or `block_write` since the hardware frame buffer block mover can be used.

Cautions

The following cautions are provided in using this driver:

1. As mentioned previously, accessing the off-screen portion of the frame buffer (using the `gescape` function) should be done with care since other processes access this region.
2. Certain `gescape` functions should be used with caution since they bypass protection mechanisms used to prevent multiple processes from interfering with each other. For example, since the hardware resources can only be rationally used by one graphics process at a time, the driver activates a semaphore and locks the device before doing any output. This ensures, for example, that process A will not change the replacement rule while process B is in the middle of filling a polygon. It also prevents the terminal (`tty`) driver from overwriting any graphics processes that are outputting to the device. The driver unlocks the device when finished processing output. Some of the `gescape` functions listed in the appendix allow the user to change this locking mechanism but should be used with *great caution*.

FINAL TRIM SIZE : 7.5 in x 9.0 in

13

MH, C+, CH, CHX, VRX Color: (Second Generation Wireframe Graphics)

Overview

This chapter describes the second generation of Hewlett-Packard's bit-mapped wireframe graphics devices. These graphics interfaces are supported on Series 300, 400, and 800 systems as described below. (Note that none of these subsystems are supported on the Series 600 or 700 computers.) All devices in this family operate at a refresh rate of 60 Hz. They do not support advanced rendering features such as hidden surface removal (via Z-buffer), smooth shading, modeling light sources, or surface property modeling.

Table 13-1. The Second Generation Wireframe Family

Device Name	Product Number	Resolution	Comments
Monochrome High Res (MH)	HP 98548A	1280×1024	
Medium Res Color (C+)	HP 98549A	1024×768	also on Model 319C
High Res Color (CH)	HP 98550A	1280×1024	
VRX Color	HP A1416A	1280×1024	same as HP 98550A
Integer Accelerator (CHX)	HP 98556A	n/a	accelerator for HP 98549A and 98550A

The second generation wireframe family includes special hardware support for the following operations:

- Copying blocks of pixels within the frame buffer.
- Area fill of polygons.
- Independent write-enable and display-enable of frame buffer planes.
- Pixel replication.

- Blinking planes.
- 3-operand raster combinations with 16×16 pixel tiling mask.
- Bit per pixel block transfers.
- Line, rectangle, and circle drawing.

Device Architecture

HP 98548A (MH)

The HP 98548A is a monochrome display subsystem supported on some Series 300 systems. (It is not supported on Series 400 or 800 systems.)

The frame buffer consists of a single (image) plane. (There is no overlay.) There is a 2-entry hardware color map associated with the image plane. This color map is writeable; each cell can hold an 8-bit grayscale value. (Note: some graphics libraries may not support altering the contents of this color map. See the appropriate subsections on device-dependencies later in this chapter.)

The screen resolution is 1280×1024 pixels. The entire frame buffer is 2048×1024 pixels. Offscreen memory is to the right of the displayed pixels.

There is no special hardware support for cursors (tracking echoes).

HP 98549A (C+) and Model 319C (Integrated Graphics)

The HP 98549A is a color graphics interface supported on some Series 300 systems. (It is not supported on Series 400 or 800 systems.) The HP 319C workstation includes the HP 98549A graphics subsystem integrated with the system processing unit (SPU).

The default configuration of the C+ is a 6-plane (color) device (i.e. all image planes, no overlay). This is known as 6-plane mode. It is possible to soft-configure this interface as 4 image planes and 2 overlay planes, also known as 4+2 mode. (The minor number of the device special file used to access the C+ determines which configuration is in effect.)

There are two hardware color maps. Each color map entry is 24 bits wide (8 bits each for red, green, and blue components). There is a 64-entry color map for the image planes. (Only 16 entries are available with the 4+2 configuration.) There is a separate 4-entry color map for the overlay planes.

This device does support overlay transparency (in the 4+2 mode). See the subsection below for more information.

The screen resolution is 1024×768 pixels. The entire frame buffer is 1024×1024 pixels. Offscreen memory is below the displayed pixels.

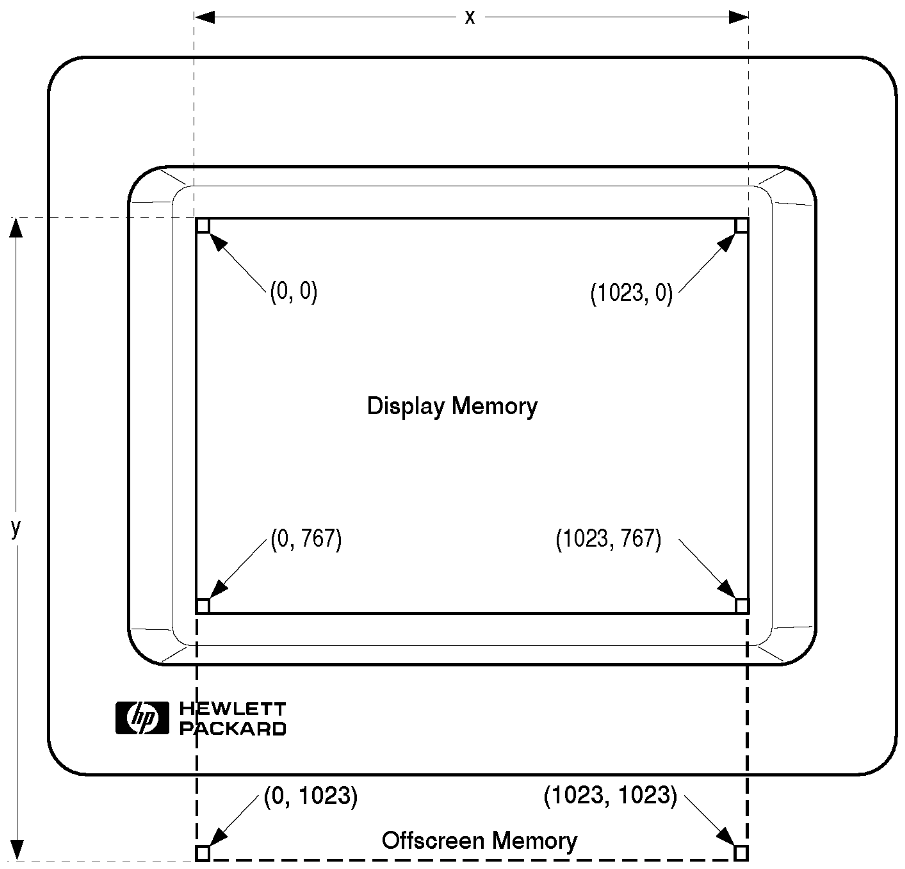

Figure 13-1. HP 98549A Physical Address Space

HP 98550A (CH) and A1416A (VRX Color)

The HP 98550A (CH) and HP A1416A (VRX Color) are graphics interfaces supported on Series 300, 400, and 800 systems.

There are 8 image planes and 2 independent overlay planes. As with the HP 98549A, there are two hardware color maps and each color map entry is 24 bits wide. There is a 256-entry color map for the image planes, and a 4-entry color map for the overlay planes.

The screen resolution is 1280×1024 pixels. The entire frame buffer is 2048×1024 pixels. Offscreen memory is to the right of the displayed pixels.

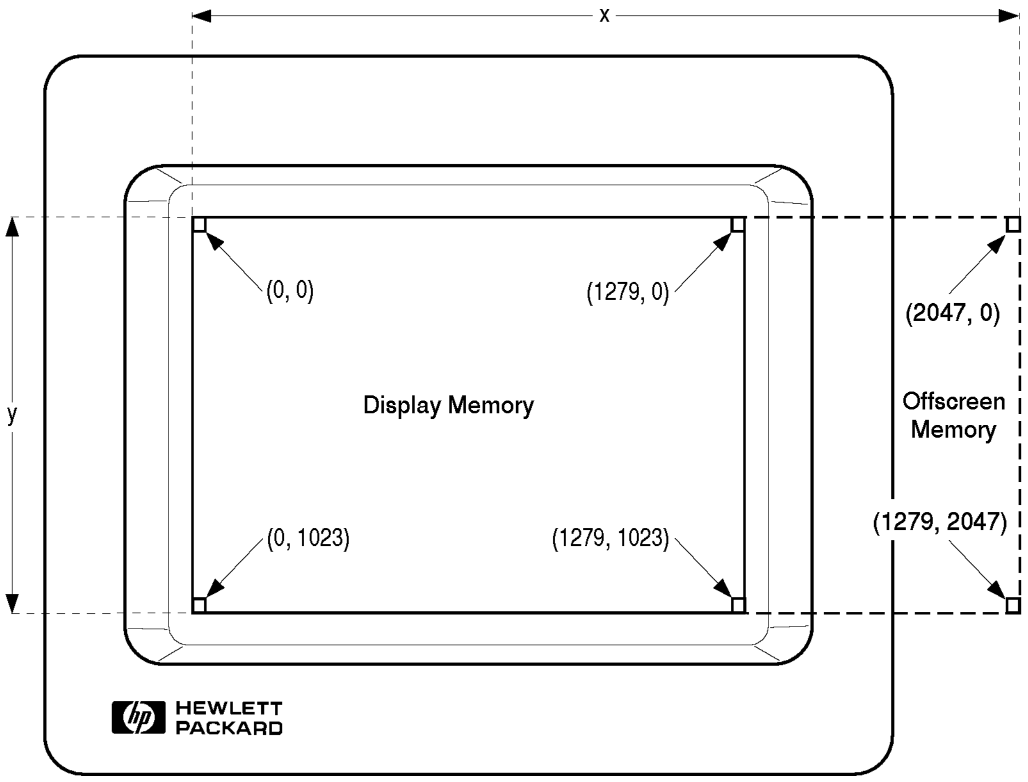

Figure 13-2. HP 98548A/98550A Physical Address Space

This device also supports overlay transparency. See the subsection below for more information. As with the 4+2 mode on the C+, it is possible to draw cursors (echoes) in the overlay planes so that they do not interfere with graphics drawn in the image planes.

HP 98556A

The HP 98556A is an optional hardware accelerator that attaches to either the HP 98549A or the HP 98550A (CHX) graphics interfaces. It supports graphics at the resolution of either of these display cards: 1024×768 for the HP 98549A and 1280×1024 for the HP 98550A. It does not attach to the VRX Color board (A1416A). It is supported on the Series 300 and Series 800.

This add-on board includes hardware and microcode support for 2-D floating point transformations, integer transformations, clipping (to rectangular boundaries), and drawing circle primitives. On HP-UX systems, the microcode for the accelerator resides in the file: */usr/lib/starbase/hp98556/gpuall.x*.

Frame Buffer Organization

Typically, the user does not need direct access to pixels in the frame buffer. However, some users may require direct frame buffer access rather than using an existing high-level graphics application programming interface (API) such as Starbase.

The frame buffer is addressed as an array of bytes, one byte for each pixel, even when there are fewer than 8 planes in the system. The least significant bits (LSB) of each pixel byte are used as indices into the corresponding color map.

For the monochrome HP 98548A, the pixel data is in the least significant bit of each byte. For the HP 98549A image planes, the pixel data is in the 6 LSBs (default, 6-plane mode) or 4 LSBs (4+2 mode) of each byte. For the HP 98550A (and VRX Color) image planes, all 8 bits of each byte are used as an index into the color map.

For the overlay planes on the HP 98550A or 98549A (in 4+2 mode), the 2 LSBs are used.

Series 800 For frame buffer access on Series 800 systems, when writing to I/O space, accesses must be on word (32-bit) boundaries. The frame buffer is mapped as one word per pixel. The pixel value is in the least significant byte of the word.

Overlay Transparency

The overlay planes index an independent 4-entry color map. Denote the four entries as colors 0, 1, 2, and 3. Colors 1, 2 and 3 are **dominant** and are always displayed for the corresponding pixels, no matter what value is stored at that location in the image planes. Color 0, however, may be made **transparent**, thus allowing the image plane pixel values to show through the overlay.

Overlay transparency affects the meaning of entry 0 in the overlay color map. If both overlay planes are display-enabled, only entry 0 of the color map is potentially transparent.

Another way to make one or more overlay planes transparent is to mask out the planes using the display-enable mask. This is equivalent to putting a zero in the corresponding bit position of the pixel data. For example, if the least-significant overlay plane is display-disabled, only entries 0 and 2 in the color map are accessed.

H-P recommends against use of this feature when accessing the graphics interface in a windowing environment. This overlay transparency feature affects the entire frame buffer, not individual pixels or windows. Therefore, it would be difficult to simulate in a general way the effect of a transparent window using this capability.

Hardware Cursor Support

There is no special hardware support for drawing cursors on second generation wireframe displays. However, it is possible to draw cursors in the overlay planes, when using a device with overlays. This techniques ensures that the cursor does not interfere with graphical objects drawn in the image planes. By using pixel value 0 as the background color index for the overlay planes and using the overlay transparency feature, the image plane graphics will "show through" the overlay planes, except for the cursor foreground pixels.

Double-Buffer Support

There is no special hardware, such as an extra bank of planes in the frame buffer, for double-buffering. The low-level display-enable and write-enable masks provide a mechanism for dividing the frame buffer into two (or more) buffers for software double-buffering. In that event, it is also necessary to initialize the color map so that it reflects the smaller number of planes in each buffer. (Note: the Starbase Graphics Library implements double-buffering using this technique.)

Multiple-Plane Bit Per Pixel Support

The supported displays provide hardware support for frame buffer access in both pixel-major and plane-major order. Pixel-major access views the frame buffer as an array of bytes, one byte per pixel, with *n* significant bits used in each byte.

Plane-major access views the frame buffer as a set of *n* planes, each consisting of a packed array of bits, one bit per pixel. This access mode allows users to transfer data quickly from one plane to another or from a data array to a single plane. (This does not alter the organization of the frame buffer, only the logical view of that organization.)

Pixel Replication

The hardware supports pixel replication in the frame buffer, using the hardware block mover capabilities.

Polygon Interior Fill Tiling

There is special hardware support for tiling fill areas with a 16×16 fill pattern.

3-Operand Raster Combinations

The supported devices include hardware support for raster operations involving different **replacement rules** (also known as **drawing modes**). These rules are, in essence, truth tables showing the results of combining two or more logical operands. They define the results of combining a source bit and a target bit. The result becomes the new value in the frame buffer.

When filling or copying an area of the frame buffer, a source **region** is combined with a target region according to the current replacement rule. This means that each bit of every pixel in the source region is combined with the corresponding bit of the target pixel according to the replacement rule.

For example, the most often used 2-operand rule is `SOURCE`, that is, the value of the source bit dominates, and is defined by the following truth table. (The term

target refers to the current pixel value in the frame buffer and **result** refers to the final pixel value in the frame buffer.)

Table 13-2. Example 2-Operand Raster Combination

Source	Target	Result
0	0	0
0	1	0
1	0	1
1	1	1

There are also **3-operand** replacement rules. For these, three logical operands can be combined; thus there are 256 different 3-operand rules. The third operand is referred to as the *mask* operand. Although it need not be considered as a logical mask, it is often used that way in common applications. (Some literature refers to the third operand as a *pattern* operand. We will not use this terminology to avoid confusion with the terms *dither pattern* or *fill pattern* which refers to the source operand used during polygon fill operations.)

Shown below is the truth table for the rule "if $\langle mask \rangle = 1$, use $\langle source \rangle$ else keep $\langle target \rangle$".

Table 13-3. Example 3-Operand Raster Combination

Mask	Source	Target	Result
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The supported displays restrict the mask operand. It is defined as a 16×16-pixel rectangle that is repeated across the entire raster in both X and Y, starting at the upper left corner. (I.e. the mask **tiles** the raster display.) This hardware tiling is tied to device coordinates, so drawing the same primitive to different locations on the screen may produce different results.

This restriction (on mask size and placement) precludes the use of the mask operand for trimming the edges of a large irregular figure, but it is convenient for imposing a pattern on the source during a `block_move` or `block_write` operation.

For example, suppose a raster rectangle is to be copied to another area on the screen using `block_move`. With the typical two-operand replacement rule, the source data is applied (unchanged) to the target area according to the current two-operand drawing mode. With the 3-operand rule, however, the mask operand could be defined as a checkerboard that actually masks the source, producing a checkerboard effect in combining source and target.

The following figure shows how this might appear, using the 3-operand rule illustrated above.

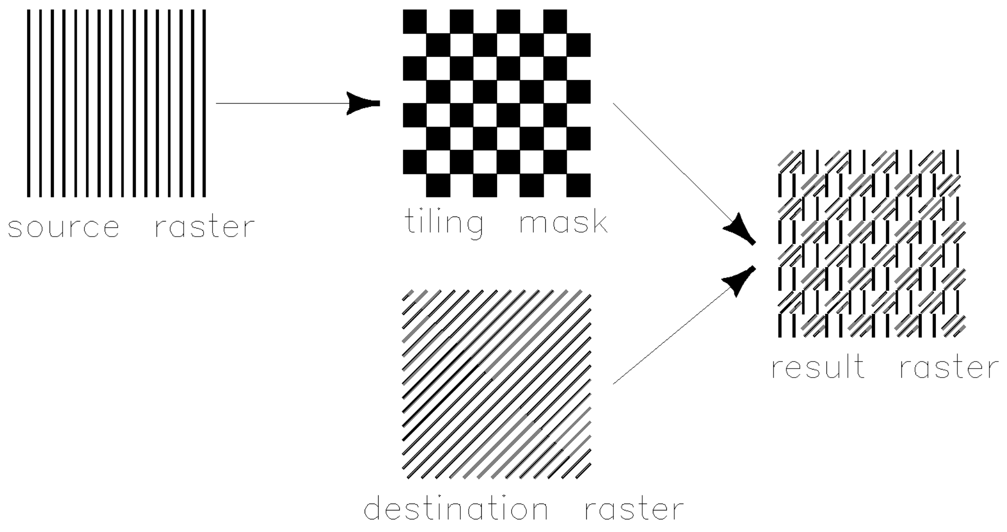

Figure 13-3. Effects of 3-Operand Replacement Rule

FINAL TRIM SIZE : 7.5 in x 9.0 in

Configuring the Graphics Hardware

Most of these interfaces include a bank of eight addressing switches on the board. (The HP 319C graphics system does not have any such switches.)

For use on Series 300 and 400 systems, the default switch setting configures the interface as the internal display (in the 24-bit DIO-I address space). The HP 319C graphics subsystem is fixed at the internal select code (0).

Setting the most significant bit of this address bank will configure the graphics interface in the 32-bit DIO-II address space. (This is not supported on the Model 320 computer.) The first four addresses in the 32-bit space are not supported.

Table 13-4. Switch Settings for the Series 300

Switch Settings	Select Code		Comments
	Dec	Hex	
00000001	0	00_x	Internal (Default)
00000010	—	—	Not Supported
⋮	⋮	⋮	⋮
01111111	—	—	Not Supported
10000000	—	—	Not Supported
10000001	—	—	Not Supported
10000010	—	—	Not Supported
10000011	—	—	Not Supported
10000100	132	84_x	External - Supported
10000101	133	85_x	External - Supported
⋮	⋮	⋮	⋮
11111111	255	FF_x	External - Supported

On some Series 800 systems the HP 98550A display board fits into an external bus convertor (HP A1020A). For the Series 800 there is only one supported switch setting for the HP 98550A: binary 1000 0100 (hex 84).

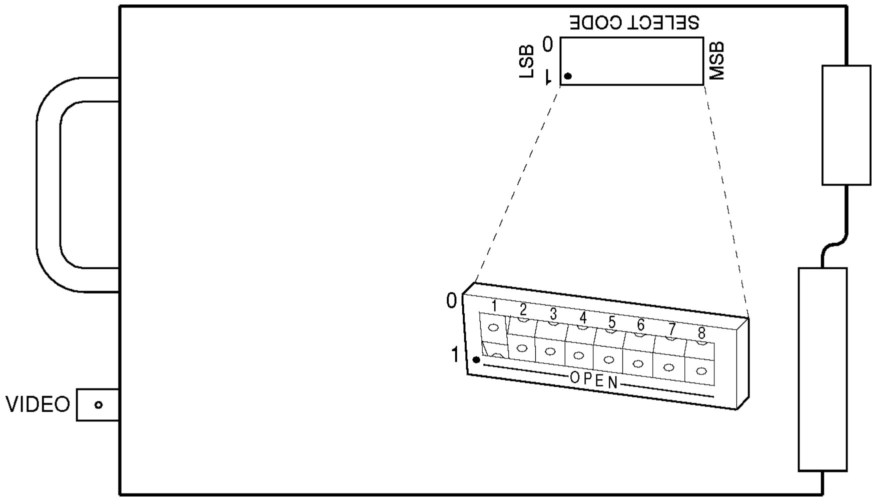

Figure 13-4. HP 98548A Switch Settings for Series 300

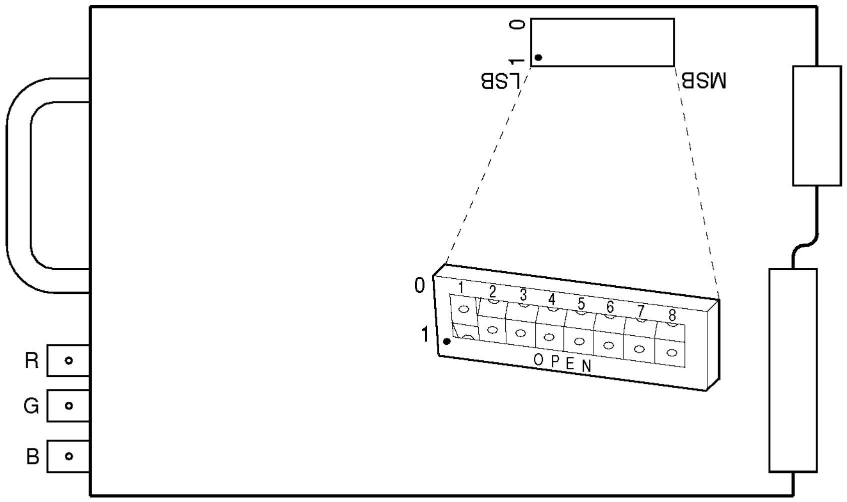

Figure 13-5. HP 98549A/98550A Switch Settings for Series 300

In order to access the HP 98556 accelerator, you must configure the HP 98549A or HP 98550A graphics interface in the external (DIO-II) address space. There are no address switches on the HP 98556 board.

Configuring the Kernel

There is no special kernel configuration required to access these devices on the Series 300 and 400.

On the Series 800, the `graph0` driver must be part of the kernel. (Consult the appropriate systems administration documentation for instructions on how to do this.)

Creating Device Special Files

The major number of a device special file identifies the kernel device driver used to access the associated physical device. Starbase and X Window System device drivers are not part of the HP-UX kernel. By convention, these are character mode devices. The major number is 12 for the Series 300 and 400. The major number is 14 for the Series 800.

The minor number specifies the address (select code for Series 300/400, or logical unit number for Series 800). When the last digit is two (2), it specifies access to the image planes. When the last digit is one (1), it specifies access to the overlay planes. For the monochrome HP 98548A, Starbase ignores the last digit of the minor number.

For the HP 98549A (and 319C built-in graphics), if the last digit of the minor number is zero (0), Starbase accesses the graphics card as a single 6-plane device. If the last digit is one (1) or two (2), it accesses the card as if it consisted of 4 image planes and 2 overlay planes (the 4+2 mode).

There is no special information in the minor number for accessing the HP 98556A integer accelerator.

Example mknod(1M) Commands

The following example creates a device special file for the monochrome HP 98548A on a Series 300 (at the internal address). The resulting device special file is also used to access the HP 98549A as a single 6-plane device.

```
mknod  /dev/crt  c  12  0x000000
```

If the interface is at an external address (select code) on the Series 300, create the device file as follows.

```
mknod  /dev/crt  c  12  0x⟨sc⟩0200
```

The next examples create device special files for accessing the HP 98549A in 4+2 mode (i.e. as a device with both 4 image planes and 2 overlay planes), again at the internal address (select code 0). Note the deviation from the conventional name for an image-plane device file.

```
mknod  /dev/i4crt  c  12  0x000002      image planes
mknod  /dev/o2crt  c  12  0x000001      overlay planes
```

The Series 300 device files for the HP 98550A at the internal address are similar to those above. (However, the names `/dev/crt` and `/dev/ocrt` are more appropriate.)

The next examples create device files for 4+2 mode of the HP 98549A when the board is at an external address.

```
mknod  /dev/i4crt  c  12  0x⟨sc⟩0202      image planes
mknod  /dev/o2crt  c  12  0x⟨sc⟩0201      overlay planes
```

As before, the Series 300 device files for the HP 98550A at an external address are similar to those for 4+2 mode on the HP 98549A.

Finally, here are examples for creating device special files for the HP 98550A on the Series 800. Rather than a select code, you must specify a logical unit number (determined when you configure the kernel) for the graphics interface.

```
mknod  /dev/crt   c  14  0x00⟨lu⟩02      image planes
mknod  /dev/ocrt  c  14  0x00⟨lu⟩01      overlay planes
```

Starbase Graphics Library

Starbase Access of the Graphics Hardware

HP 98549A

Do not open the HP 98549A simultaneously in both 6-plane mode and 4+2-plane mode. Doing so will cause indeterminate results. You may simultaneously open the 2 overlay planes and the 4 image planes using two different file descriptors. If drawing into the overlay planes, do not use the gescape `R_OVERLAY_ECHO` to move the cursor from the image planes into the overlay planes. This will interfere with the graphics operations in the overlay planes.

Frame Buffer Access

For applications that require direct access to the frame buffer, Starbase provides the `R_GET_FRAME_BUFFER` gescape that returns the virtual memory address of the start of the frame buffer in the process logical address space. (Read Appendix A for specific information about this gescape.) Frame buffer locations are then addressed relative to the returned address, in byte-per-pixel mode.

On Series 800 systems, to ensure valid direct frame buffer access, the user must precede the `R_GET_FRAME_BUFFER` gescape with the `R_LOCK_DEVICE` gescape. After completing the frame buffer access and prior to any other Starbase commands, the user must call the `R_UNLOCK_DEVICE` gescape.

The supported displays provide hardware support for frame buffer access in both plane-major and pixel-major modes. Pixel-major access views the frame buffer as an array of bytes, one byte per pixel, with *n* significant bits each byte. The normal operation of Starbase procedures `block_read` and `block_write` is to access the frame buffer in pixel-major mode.

Plane-major mode addresses the frame buffer as a set of *n* planes, each consisting of a packed array of bits, one bit per pixel. This may be used, for example, to transfer data quickly from one plane to another or from a data array to a plane. In this driver, plane-major access may be made with the `block_read` and `block_write` procedure by setting the *raw* parameter to `TRUE` and enabling *raw* mode using the `R_BIT_MODE` gescape. (Not all drivers provide this capability.) See the subsection of this chapter entitled "Starbase Calls That Are Conditionally Supported" for more information about plane-major access.

In plane-major mode, the programmer must provide pixel data storage (target for `block_read`, source for `block_write`) that is organized as follows. Data from each plane will be packed, eight pixels per byte. Each row will begin on a byte boundary (implying that the end of each row may be padded with unused bits).

The following equation shows the size (in bytes) of the rectangle specified by the *length_x* and *length_y* parameters.

$$plane\_storage = \frac{length\_x+7}{8} \times length\_y$$

Data for the next plane begins on the following byte boundary.

When reading into or writing from the packed pixel data array, Starbase will clip to the screen limits. For `block_read`, the first pixel of the source rectangle is placed in the high-order bit of the first byte for each plane region. If the `clip_rectangle` area intersects the source region (in the frame buffer), Starbase will not read the entire region indicated by the input parameters. Starbase performs analogous unpacking and clipping operations for `block_write`.

The `R_BIT_MASK` and `GR2D_PLANE_MASK` gescapes define a bit mask that selects the planes to read or write. The default (i.e. initial) value of this mask is (binary) 0000 0001, indicating access to only the low-order plane. See the Appendix for details on how to use these gescapes.

The plane-selection mask defines a set of planes to be stored as consecutive packed plane regions in the pixel data array. This ensures efficient use of storage for the data, but also presents the potential for addressing violations or unexpected results.

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

HP 98549A Caution is necessary if the HP 98549 is opened in the 4+2-plane mode, because the frame buffer address returned for the 4 image planes is the same as the address for the 2 overlay planes. To ensure that you only access the planes opened, the `R_LOCK_DEVICE` gescape (using the file descriptor for the appropriate planes) should be used to lock the device before reading or changing the frame buffer.

Note that your program must shift each data byte to the left by four bits in order to write it to the overlay planes. Use the `R_LOCK_DEVICE` gescape to unlock the device after the access.

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) 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 ($1023 \times 2048 + 1279$).

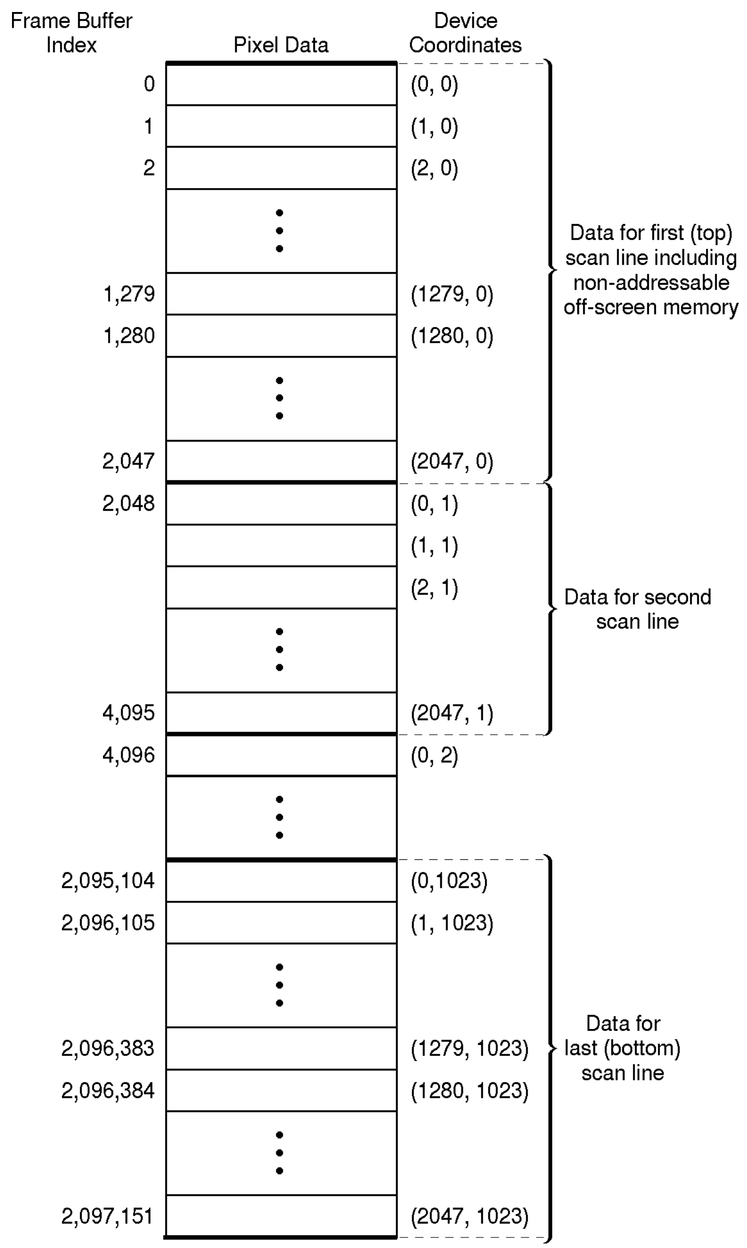

Figure 13-6. Frame Buffer to VM Mapping for HP 98548A/98550A

Offscreen Memory Usage

Both the *hp98550* and *hp98556* drivers allocate a portion of offscreen memory for each `gopen` to hold fill patterns and raster echo definitions.

The first 32 lines of offscreen frame buffer memory are reserved. The remainder of offscreen memory is available for raster font bitmaps, or Starbase raster echo storage.

These drivers allocate 64 lines of offscreen memory each time it needs space for a raster echo. The storage needed by font optimization varies with the font size. In general, offscreen frame buffer memory is allocated by the system from top to bottom (i.e., from low offsets to high). Review the descriptions of the gescapes `R_OFFSCREEN_ALLOC` and `R_OFFSCREEN_FREE` in the Appendix to this manual.

Hewlett-Packard advises against the use of offscreen memory while the X Window System is running.

Multiple-Plane Bit Per Pixel Support

When `block_read` or `block_write` is used with the *raw* parameter `TRUE`, and *raw* mode is enabled by the `R_BIT_MODE` gescape, the driver supports bit per pixel frame buffer access to single planes.

The gescape `R_BIT_MASK` defines a plane mask to the driver and is used for bit per pixel access. As in other device drivers, only the plane corresponding to the highest bit set in the mask is transferred. This gescape is supported for retained rasters; i.e. the correct data is returned from the retained raster for those parts of the window that are obscured.

The gescape `GR2D_PLANE_MASK` defines a mask that allows multiple planes to be read or written. This gescape is not supported in retained rasters. It returns the correct data from the visible portions, but not from the obscured portions.

Pixel Replication

`GR2D_REPLICATE` triggers operations to do pixel replication in the frame buffer. This is a complicated operation involving multiple hardware `block_move` operations; the retained raster will not be affected.

Polygon Interior Fill Tiling

GR2D_FILL_PATTERN sets a 16×16 fill pattern used by the driver as a source to fill polygons until the fill pattern is redefined, either by a call to `fill_color` or `fill_color_index` or to the gescapes R_DEF_FILL_PAT or GR2D_FILL_PATTERN. The retained raster only supports the R_DEF_FILL_PAT 4×4 pattern. The `pattern_define` routine for Starbase is recommended instead of this gescape.

3-Operand Raster Combinations

Access to the hardware capability is provided through three gescape operations:

- GR2D_DEF_MASK defines the 16×16 mask operand
- GR2D_MASK_RULE defines the 3-operand rule
- GR2D_MASK_ENABLE enables or disables the 3-operand combination

When the feature is disabled, the current 2-operand rule is in effect. When enabled, the mask operand and 3-operand rule may be applied to `block_write` and `block_move` operations, or to these plus raster text operations.

Since these gescapes alter the rule and pattern used for `block_write` and `block_move`, the retained raster will be affected during later raster operations and the results will not be consistent with what appears on the screen.

Note For a discussion of replacement rules, review the "Drawing Modes" section of the "Frame Buffer Control and Raster Operations" chapter of the *Starbase Graphics Techniques* manual.

Overlay Transparency

There is a Starbase gescape, GR2D_OVERLAY_TRANSPARENT, to mark overlay color index 0 as dominant. This ensures that all overlay plane colors will be displayed, thus entirely obscuring the image planes. The default mode is for overlay color index 0 to be transparent.

Device Coordinate Addressing

For device coordinate operations, the upper-left corner of the screen is location (0, 0). X-axis values increase from left to right and Y-axis values increase from top to bottom of the screen. The lower-right corner of the display is (1279, 1023) for the high resolution display cards (MH, CH, VRX Color) and (1023, 767) for the medium resolution cards (C+, HP 319C).

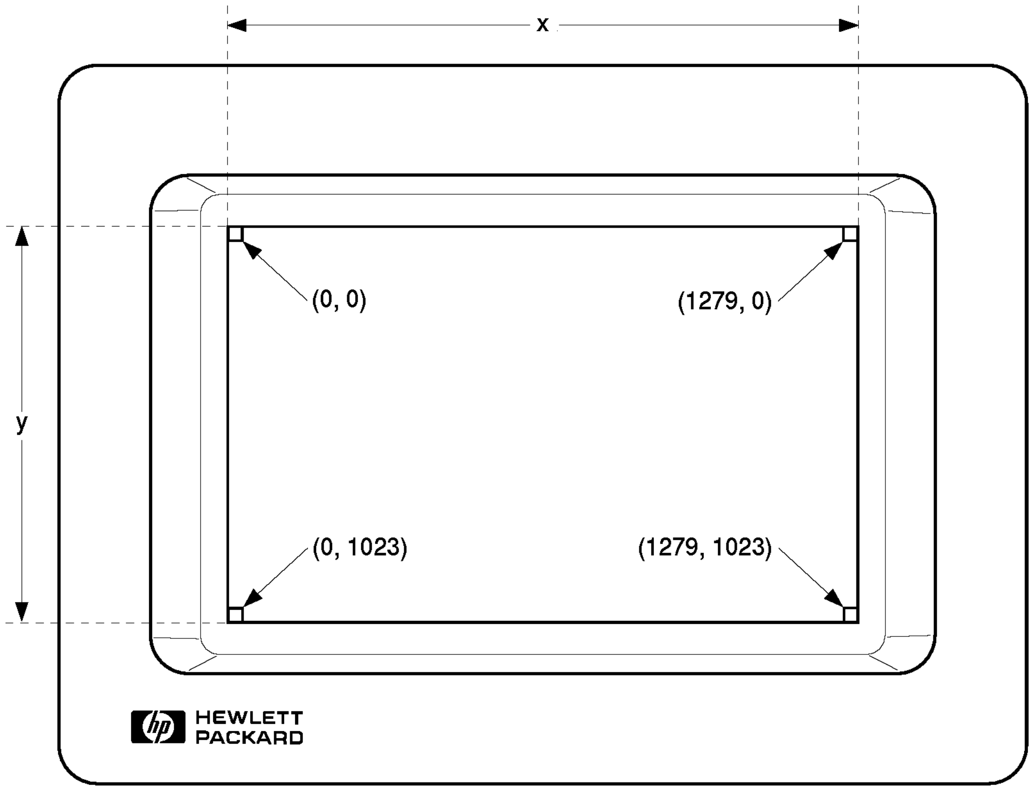

Figure 13-7. Device Coordinates for the HP 98548A/98550A

Retained Raster

The HP 98556 driver does not support retained windows. Output to obscured parts of a retained window will not affect the retained raster. In order to be compatible with older applications that require retained rasters, the HP 98556 driver behaves as follows when it is used to `gopen` a retained window:

1. If the HP 98550 driver is also linked into the user program, Starbase will substitute the HP 98550 driver for the HP 98556 driver during `gopen`. A Starbase warning of "`Driver name substituted on gopen`" will be generated during the `gopen`.

2. If the HP 98550 driver is not linked into the user program, Starbase will use the HP 98556 driver. A Starbase warning of "`Driver doesn't support retained rasters`" will be generated during the `gopen`. The `gopen` will succeed, but remember that output to the obscured parts of a window will not be saved in the retained raster.

Starbase Echoes

Only one Starbase echo is supported in a window by the *hp98556* driver. When a window is opened multiple times by the *hp98556* driver, only one of these opens should specify a Starbase echo because the *hp98556* driver can "pick up" only one Starbase echo and one X11 cursor.

When a window is opened twice by the *hp98556* driver and each open specifies a Starbase echo, the first invocation of the driver will not be able to pick up the echo generated by the second invocation of the driver.

The maximum size allowed for a raster echo is 64×64 pixels. The default drawing mode for the raster echo is 7 (logical *OR*). By default, all echo types are written to the open planes. The location of raster and non-raster echoes may be changed by using the `R_OVERLAY_ECHO` gescape.

Starbase echoes may reside in either the overlay or the image planes. The two logical devices may be opened simultaneously, but moving the image planes cursor into the overlay planes may interfere with other graphics drawn in those planes.

Starbase Device Drivers

There are two device drivers especially designed for use with the second generation wireframe family. the *hp300h* driver (intended for first generation wireframe devices) also works with the HP 98549A, but there are limitations in terms of functionality and performance. The table below describes the supported drivers.

Table 13-5. Starbase Drivers for Second Generation Wireframe

Driver Name	Driver File Names	Comments
hp98550	*libdd98550.a* *libdd98550.sl*	unaccelerated driver
hp98556	*libdd98556.a* *libdd98556.sl*	accelerated driver
hp300h	*libdd300h.a* *libdd300h.sl*	for use with HP 98549A only

You can use the *hp300h* driver with the HP 98549A display board and HP 319C display, but only if the graphics interface is configured as the internal display (i.e. at select code 0). The *hp300h* driver accesses the HP 98549A as if it were the HP 98547A graphics card (part of the 1st generation wireframe family). In most cases the *hp300h* driver will be slower at drawing than the *hp98550* driver.

Caution: do not use the *hp300h* and *hp98550* drivers simultaneously to access the same HP 98549A card. This includes different applications running at the same time (each using a different driver) and also any of the windowing systems supported on this device. These drivers manage offscreen frame buffer memory in different ways and so will interfere with each other.

The HP 98556 device driver is used to interface the Starbase Graphics Library with the HP 98556A accelerator when attached to the HP 98549A or the HP 98550A. This configuration allows for use of multiple high speed windows using either HP Windows/9000 or the X11 Window system.

The HP 98556 driver should be used when speed is very important and integer or DC graphics operations are performed. When speed in graphics operation performance is not as important, the HP 98550 driver should be used. The HP 98556 driver only supports 31 simultaneous `gopens`. Any additional `gopens` must be to the HP 98550 driver.

Linking Starbase Programs

Shared Libraries

By default, the HP-UX link editor `ld(1)` will link with the Starbase shared library version of the device drivers. The driver(s) will be dynamically loaded at run time.

What follows is an example of how to link a Starbase graphics program to display output in an X11 window. This example compiles and then links a C program with the shared library driver:

```
$ cc -I/usr/include/X11R5/x11  -L /usr/lib/X11R5\
     -lXwindow  -lsb  -lXhp11  -lX11  -lm  -ldld\
     example.c  -o example
```

or with FORTRAN 77 use,

```
$ f77 -Wl,-L/usr/lib/X11R5 -lXwindow -lsb -lXhp11\
      -lX11  -lm  -ldld -o example example.f
```

or with Pascal use,

```
$ pc -Wl,-L/usr/lib/X11R5 -lXwindow -lsb -lXhp11\
     -lX11 -lm -ldld -o example example.p
```

Here is another example, one that shows how to link a program that accesses the graphics device in raw mode:

```
$ cc example.c -lsb -lm -ldld -o example

$ f77 example.f -lsb -ldld -o example

$ pc example.p -lsb -ldld -o example
```

Upon device initialization the proper driver will be loaded. Refer to the "Starbase Initialization" subsection of this chapter for details.

Archive Libraries

It is possible to link with archive versions of the Starbase libraries by either referencing the library by the absolute path name `/usr/lib/libdd98550.a` or an appropriate relative path name. Alternatively, use the `-l` option as in `-ldd98550` with the `LDOPTS` environment variable set to `-a archive`.

The reason for using the `LDOPTS` environment variable is that the `-l` option will look for a shared library driver first and then look for the archive driver if no shared version was found. By exporting the `LDOPTS` variable as specified above, the `-l` option will only look for archive drivers. For more information about linking with shared or archive libraries, refer to the *Programming on HP-UX* manual.

For example: to compile and link a program that will access the display in the X Window System, use the following commands. (We assume that the reader is using `ksh`(1).)

```
      $ export  LDOPTS="-a archive"
```

For C, this is followed by:

```
   $ cc -I/usr/include/X11R5/x11 -L /usr/lib/X11R5\
        -ldd98550  -lXwindow -lsb -lXhp11 -lX11 -lm\
        -ldld -o example example.c
```

Or, for FORTRAN 77, use:

```
   $ f77 -Wl,-L/usr/lib/X11R5 -ldd98550 -lXwindow\
         -lsb -lXhp11 -lX11 -lm -ldld -o example example.f
```

And for Pascal programs, use this:

```
   $ pc -Wl,-L/usr/lib/X11R5 -ldd98550 -lXwindow -lsb -lXhp11\
        -lX11 -lm -ldld -o example example.p
```

Another example shows how to compile and link a program that uses raw-mode access to the unaccelerated HP 98550A device:

```
   $ cc example.c -ldd98550 -lsb1 -lsb2 -o example

   $ f77 example.f -ldd98550 -lsb1 -lsb2 -o example

   $ pc example.p -ldd98550 -lsb1 -lsb2 -o example
```

Starbase Initialization via gopen(3G)

Parameters for gopen

The `gopen` call requires four input arguments: *path*, *kind*, *driver*, and *mode*.

path	This is the name of a device special file created by the `mknod` command as specified in the System Administration section of this chapter, e.g. `/dev/crt`. It can also be the name of a pseudo-TTY device file created by the `xwcreate`(1X) command, or a special `gopen` string returned by the Starbase `make_X11_gopen_string`(3G) call.
kind	This argument indicates the type of I/O access — input only, output only, or both input and output — via this `gopen` file descriptor. For raw mode device access, *kind* must be the constant `OUTDEV` (defined in the *starbase*.h* header files). For a graphics window, it may be either `OUTDEV` or `OUTINDEV`.
driver	This is the device driver name, specified as a C-style string.

The *driver* name may be the constant `NULL` (i.e. a null string), in which case Starbase will inquire the device type at run-time and determine the appropriate device driver. To specify this use the following syntax for various programming languages:

`NULL`	for C (defined in *stdio.h*)
`char(0)`	for FORTRAN 77
`''`	for Pascal

In this case, if the application is linked to use Starbase shared libraries, Starbase will use the dynamic loader (`-ldld`) to link to the appropriate driver at run-time.

By default, Starbase selects the accelerated driver for a device. You can override this selection by including either the `ACCELERATED` or the `UNACCELERATED` flag in the *mode* argument to `gopen`.

Alternatively, the *driver* argument may be a character string indicating the device name. This overrides the `ACCELERATED` and `UNACCELERATED` flags. The Starbase device drivers for this family of devices accept the following driver names:

Table 13-6. Valid *driver* Name Strings

driver String	Selected Driver

`hp98548`
`hp98549`
`hp98550`
	unaccelerated driver

`hp98556`	accelerated driver
`hp300h`	*hp300h* driver (unaccelerated)

The driver will correctly open any of the supported displays if one of these strings is used as the driver name. Call the `inquire_id`(3G) routine to determine which specific display model was identified.

mode This argument consists of several flag bits **or**-ed together. The following list describes flag bits and their effects with these drivers. Other flag bits operate as defined on the `gopen`(3G) page of the *Starbase Reference* manual. For a description of `gopen` actions when accessing an X window, refer to *Starbase Graphics Techniques*.

`0` (zero)

1. Open the device, but do not alter its state in any way.
2. Initialize the software color table with the current hardware color map values. (The minor number of the device special file indicates the number of planes, and thus the number of color map entries, to read.)

`INIT` Open the device and initialize it as follows:

1. Clear the frame buffer (image planes or overlay planes, depending on the *path* argument) by setting all pixels to the default background color index (zero).
2. Reset the color map to the Starbase defaults.
3. Enable all planes in the frame buffer for reading (display) and writing.
4. If opening the image planes, configure the overlay planes so that zero values are transparent.

`RESET_DEVICE` Open the device and initialize it as follows:

1. Reset the hardware to its boot-up state.
2. Clear the image planes. (I.e. set all pixels to zero.)
3. Reset the image plane color map to the Starbase defaults.
4. Clear the overlay planes.
5. Reset the overlay plane color map to the Starbase defaults.
6. Enable all planes in the frame buffer for reading (display) and writing.
7. If opening the image planes, configure the overlay planes so that zero values are transparent.
8. (For the HP 98556A only) Download microcode for the accelerator.

`SPOOLED`	These devices do not support spooled output. (This is true for all HP bit-map raster devices.)
`MODEL_XFORM`	These devices do not support smooth shading. However, this flag, modifies the way that Starbase performs matrix stack and transformation operations.

Example gopen Calls

The following examples show how to open and initialize the HP 98550A (or VRX Color) device for output. Note that most of the examples use the `NULL` device driver specification. This method provides greater portability of Starbase applications from one graphics interface to another. (See the chapter on "Developing a Starbase Application" for more information on this subject.)

For C programs:

```
sb_display = gopen("/dev/crt", OUTDEV, NULL,
                   UNACCELERATED|INIT);
```

or (with explicit naming of the device driver):

```
sb_display = gopen("/dev/crt", OUTDEV, "hp98550", INIT);
```

For FORTRAN 77 programs:

```
DEVICE = '/dev/crt'//char(0)
```

```
SB_DSP = gopen(DEVICE, OUTDEV, char(0), INIT)
```

For Pascal programs:

```
sb_display := gopen('/dev/crt', OUTDEV, '', INIT);
```

Starbase Calls That Are Not Supported

The following Starbase procedures are not supported by either the unaccelerated (*hp98550*) or the accelerated (*hp98556*) driver. Calling any of these procedures will have no effect (i.e. they are no-ops).

frame buffer control	`bank_switch`
color map manipulation	`shade_range`

polygon attributes	`backface_control` (obsolete)
	`bf_control`
	`bf_fill_color`
	`bf_interior_style`
	`bf_perimeter_color`
	`bf_perimeter_repeat_length`
	`bf_perimeter_type`

spline curves/surfaces	`define_trimming_curve`

model clipping/capping	`set_capping_planes`
	`set_model_clip_indicator`
	`set_model_clip_volume`

depth cueing	`depth_cue`
	`depth_cue_color`
	`depth_cue_range`

hidden surface removal	`hidden_surface` `zbuffer_switch`

light/surface modeling	`bf_surface_coefficients`
	`bf_surface_model`
	`light_ambient`
	`light_attenuation`
	`light_model`
	`light_source`
	`light_switch`
	`surface_coefficients`
	`surface_model`
	`viewpoint`

anti-aliasing	`line_filter` `perimeter_filter`
alpha transparency	`alpha_transparency` `bf_alpha_transparency`

texture mapping	`bf_texture_index`
	`define_texture`
	`texture_index`
	`texture_viewport`
	`texture_window`

contouring	`contour_enable` `define_contour_table`
deformation animation	`deformation_mode`

Starbase Calls That Are Conditionally Supported

The following Starbase procedures are conditionally supported by both the unaccelerated (*hp98550*) and the accelerated (*hp98556*) driver, as described below.

`block_read`, `block_write`	The default storage organization for raster transfer operations is pixel-major order. If the `raw` parameter is `TRUE`, the driver will perform `block_read` and `block_write` operations as plane-major reads and writes. This is enabled by the `R_BIT_MODE` gescape.
`interior_style`	The `INT_OUTLINE` and `INT_POINT` area fill styles are not supported.

`shade_mode`	This routine will set the color map mode (`CMAP_NORMAL`, `CMAP_MONOTONIC`, `CMAP_FULL`), but neither device driver for this family supports smooth shading.
`text_precision`	Only `STROKE_TEXT` precision is supported. (Use the FA/FM libraries or X11 text calls for an alternative text mode: raster text.)
`vertex_format`	The ⟨*use*⟩ parameter indicates whether there is additional data per vertex (which is used in color calculations). This argument must be zero; Starbase will ignore the extra coordinate data.
`*_with_data*`	The following calls designate that there is additional data per vertex used for special functionality: `partial_polygon_with_data3d` `polygon_with_data3d` `polyhedron_with_data` `polyline_with_data3d` `polymarker_with_data3d` `quadrilateral_mesh_with_data` `triangle_strip_with-data` The second generation wireframe drivers do not support contouring.

The following Starbase procedures are conditionally supported by the accelerated (*hp98556*) driver, as described below. These limitations do not apply to the unaccelerated (*hp98550*) driver.

`arc`, `circle`, `ellipse`	Starbase renders these primitives by approximating them with multi-sided polygons or polylines. These polygons in turn are limited to 255 vertices. It may be necessary to reduce the number of sides in the polygon approximation using the `curve_resolution` call.
`gopen`	The HP 98556 hardware supports context information for up to 31 simultaneous `gopen` operations. (This limit applies to all concurrently running processes that access a specific display card.) If you need to `gopen` a device more than 31

	times, use the unaccelerated (*hp98550*) driver for operations that do not require the accelerator.
`polygon*`	Polygons of up to 255 vertices are supported. For polygons with more than 255 vertices, only the first 255 vertices are displayed.

Starbase Gescapes That Are Supported

Both the unaccelerated (*hp98550*) and accelerated (*hp98556*) drivers support the following gescapes. In a few cases these drivers support a given gescape with limitations. (See the Appendix for more information about each gescape.)

`BLINK_PLANES`	Blink display planes at hardware-defined rate.
`GR2D_DEF_MASK`	Define mask for 3-operand replacement rules.
`GR2D_FILL_PATTERN`	Define 16×16 dither and fill pattern.
`GR2D_MASK_ENABLE`	Enable 3-operand replacement rules.
`GR2D_MASK_RULE`	Specify 3-operand replacement rule.
`GR2D_OVERLAY_TRANSPARENT`	Enable/disable transparency of zero-value pixels in overlay planes. The second generation wireframe family does not support transparency on a per window basis. Therefore, use of this gescape while running a windowing system may produce unexpected (and undesirable) results.
`GR2D_PLANE_MASK`	Select planes to be effected by bit per pixel block transfers.
`GR2D_REPLICATE`	Pixel zoom (with 2×, 4×, 8×, or 16× pixel expansion factor)
`R_BIT_MASK`	Select plane for reading and writing bit blocks.
`R_BIT_MODE`	Specify data format for bit per pixel block transfer operations.
`R_DEF_ECHO_TRANS`	Define a transparency mask for user-defined raster echoes. These drivers will accept a mask of up to 16×16 pixels. If the current raster echo is larger than this size, the gescape will have no effect.

R_DEF_FILL_PAT	Define 4×4 pixel dither cell for area fill.
R_FULL_FRAME_BUFFER	Enable access to entire frame buffer, including offscreen memory.
R_GET_FRAME_BUFFER	Read addresses for the frame buffer and control space of a graphics interface.
R_LOCK_DEVICE	Lock device to prevent access by other processes.
R_OFFSCREEN_ALLOC	Allocate offscreen frame buffer memory.
R_OFFSCREEN_FREE	Deallocate offscreen frame buffer memory.
R_OVERLAY_ECHO	Specify location of cursors (in overlay or image planes).
R_UNLOCK_DEVICE	Unlock device to allow access by other processes.
READ_COLOR_MAP	Read the hardware color map associated with the indicated Starbase file descriptor.
SWITCH_SEMAPHORE	Enable/disable semaphore control of access to the device.

The following gescape is supported only by the accelerated (*hp98556*) driver:

GR2D_CONVEX_POLYGONS	Increase polygon drawing speed by switching to algorithm that works only for convex polygons.

Environment Variables

There are no supported environment variables that effect the operation of Starbase on these devices.

However, there are certain environment variables that affect Starbase in general. Consult the *Starbase Graphics Techniques* for information on the GRM_SIZE and SB_DISPLAY_ADDR environment variables.

Support of Other Graphics Libraries

Fast Alpha / Font Manager

Both device drivers (*hp98550*, *hp98556*) support raster text calls from the Fast Alpha (FA) and Font Manager (FM) libraries. (Note that this is the only case where an accelerated driver supports FA/FM. See the *Fast Alpha/Font Manager Programmer's Manual* for additional information about these libraries.

To ensure portability of applications that display raster text, use the X Window System text calls. (See the *Programming with Xlib* manual for information on how these features.)

Graphical Kernel System (GKS)

The (obsolete, now in support life) HP GKS product uses the Starbase programming library and device drivers to access graphics hardware. HP GKS is supported with both drivers (*hp98550*, *hp98556*). For more information on the use of HP GKS with these devices, consult the *HP GKS Device Drivers Manual*.

Starbase Radiosity and Ray Tracing

The Starbase Radiosity and Ray Tracing (SBRR) library is not supported on these devices. The hardware does not support the necessary advanced rendering capabilities.

X Window System

The X Window System is supported in various configurations and modes on the devices supported by the HP 98550 driver. See the *Starbase Graphics Techniques* for a complete discussion of X Window support.

How X Windows Uses the Graphics Interface

The X Window System automatically accesses the HP 98549A in a way compatible with the *hp98550* driver, not with the *hp300h* driver. X will only use the HP 98549A as a 6-plane device. (There is no support for the 4+2 mode.)

The HP 98556A driver supports hardware accelerated windows. Clip information for each window that is `gopen`ed with the HP 98556 driver is downloaded to the HP 98556A accelerator by the window manager or server. This enables the HP 98556A to clip output to the visible parts of the window. The clip information for each window consists of a list of visible rectangles. Performance of output to an obscured window will degrade linearly as a function of the number of visible rectangles.

The HP 98556 driver imposes a limit of 32 simultaneous gopen's of the HP 98556 device. This limit also applies to the window system. The window manager will `gopen` the HP 98556A, allowing up to 31 windows to use the HP 98556 driver. Once a window is `gopen`ed with the HP 98556 driver, it counts against this 31 window limit until the window is closed.

If an open of the HP 98556A is performed when 31 open commands of the HP 98556A are currently active, a Starbase error is generated and the open command will fail. When one of the previous HP 98556A opens is closed, the first open command can be tried again.

When the window manager or X server uses the HP 98556 driver, graphics windows can be `gopen`ed with both the HP 98550 and HP 98556 drivers. Windows that are `gopen`ed with the HP 98550 driver are not counted against the limit of 31 accelerated windows. (The terminal emulator windows do not access the HP 98556A accelerator even when it is present.)

When a graphics image is drawn to the obscured portion of the window, only the visible parts of the window are drawn to; the obscured parts are ignored. In the X Windows System, the application should handle exposure events.

The following subsections describe how Starbase utilizes the underlying graphics hardware in the second generation wireframe family.

The following subsections describe how the Hewlett-Packard implementation of the X Window System (Version 11) uses the underlying graphics hardware for the second generation wireframe devices.

Accessing Frame Buffer Memory

Offscreen Memory

The first 32 lines of offscreen frame buffer memory are reserved for use by the X display server.

The X Window System uses offscreen for temporary and client pixmaps.

X Cursors (and Starbase Echoes)

The following list shows default positions where the X11 cursor and Starbase echo (called cursor and echo, respectively) reside for each of the supported X11 server modes.

- Overlay Mode

  Echo placed in opened image or overlay planes. Cursor resides in overlay plane.
- Image Mode

  Echo and cursor share the image planes.
- Stacked Screen Mode

  If overlay-plane window is opened, echo and cursor share overlay planes. If image-plane window is opened, echo and cursor share image planes.

Support for Backing Store and Save Under

Consider a graphics window that is partially obscured by another window. What happens when you try to draw graphics to the part of the window that is obscured? There are two options:

- Draw only to the visible parts of the window and ignore any parts that are obscured. In X Windows System, an application will receive and handle exposure events.

- For those parts of the window that are obscured, draw the image to *memory* instead of the frame buffer. When the affected portion of the window is made visible (unobscured), the window system updates the display with the appropriate graphical data from memory. Graphical data stored (retained) in memory is called a **retained raster** or **backing store**.

The X Window System supports both options. Support for retained rasters is provided by `/usr/lib/libddbyte.a` (called the byte driver). The graphics window must be created with a retained raster. Linking the byte driver allows Starbase to draw images to memory for obscured parts of the window and allows the window system to update the screen from memory if previously obscured parts of the window are made visible. See the *Starbase Graphics Techniques* for more information on retained rasters in X Windows.

The `/usr/lib/libddbit.a` (bit driver) is supported only on the HP 98548A.

Retained Raster Support

In general, those Starbase operations that draw to the display are also supported as a retained raster by the byte driver (`/usr/lib/libddbyte.a`). There are, however, some exceptions to this. These exceptions all involve use of a gescape to access device-dependent features.

When the gescapes listed below are used with a retained graphics window, they will have the desired effect for the visible portion of the window but may cause the retained raster for obscured parts to be altered in inconsistent ways. The features involved (along with the names of the affected gescapes) are listed below. For more details on the gescapes, refer to the section of this chapter entitled "Starbase Gescapes That Are Supported".

Note Because the gescapes are device-dependent, the exceptions discussed below may be removed in future drivers. Also, if the exceptions to retained raster support discussed below prove troublesome in your application, Hewlett-Packard recommendds that you consider not using retained rasters but instead detect window events and repaint the window when a previously obscured portion of a window is made visible.

Transparent Windows

The HP implementation of the X Window System does not support combined mode servers on any of the second generation wireframe devices. Therefore it is not possible to implement **floating text** windows in the overlay planes. Since the overlay transparency mechanism affects the entire frame buffer, it would be difficult to simulate this feature in any event.

Supported Server Modes

Supported Visuals

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

How to Read the Supported Visuals Tables

The tables of Supported "X" Windows Visuals contain information for programmers using either Xlib graphics or Starbase. These tables list what depths of windows and colormap access modes are supported for a given graphics device. They also indicate whether or not backing store (aka "retained raster") is available for a given visual.

You can use these tables to decipher the contents of the *X*screens* file on your system. The first two columns in the table show information that may be in the *X*screens* file. Look up the *depth=* specification in the first column. If there is no *doublebuffer* keyword in the file, look up *No* in the second column. Otherwise, look up *Yes*. The other entries in that row will tell you information about supported visual classes and backing store support.

You can also use the tables to determine what to put in the *X*screens* file in order to make a given visual available. For example, suppose that you want 8-plane windows with two buffers for double-buffering in Starbase. Look for "8/8" in the table to see if this type of visual is supported. If it is, then you will need to specify "doublebuffer" in the *X*screens* file. You will find the "depth" specification as the first entry in that row of the table.

The supported X11 server modes are: Overlay, Image, and Stacked Screen. (The availability of overlay and stacked screens modes, of course, depends on whether or not there are overlay planes on the hardware in question.)

Table 13-7. Windows on HP 98548A

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
1	No	GrayScale	●	●	

Table 13-8. Windows on HP 98549A (6-Plane Mode Only)

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
6	No	PseudoColor	●	●	
	Yes (3/3)	PseudoColor	●	●	

Table 13-9. Windows in HP 98550A Overlay Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
2	No	StaticGray	●	●	one color reserved for transparency

Table 13-10. Windows in HP 98550A Image Planes

Contents of X0screens		Visual Class	Backing Store		Comments
depth	**doublebuffer?**	**Xlib**	**Xlib**	**SGL**	
8	No	PseudoColor	●	●	
	Yes (4/4)	PseudoColor	●	●	

The Internal Terminal Emulator

How the ITE Uses the Graphics Interface

On the HP 98548A, the Internal Terminal Emulator (ITE) operates in the single-plane frame buffer. Use of color text escape sequences will have no effect on this graphics interface.

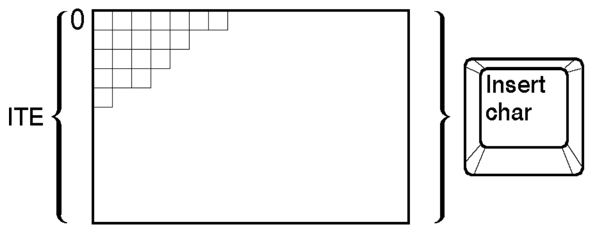

Figure 13-8. ITE on the HP 98548A (MH) Display

On the HP 98549A (and HP 319C), the ITE accesses the frame buffer as a single 6-plane device. On the HP 98550A, the ITE accesses the image planes. On these devices the ITE uses three planes thus supporting eight different colors of text. The ITE uses the low-order planes (i.e. those planes corresponding to the least significant bits of each pixel) in the frame buffer.

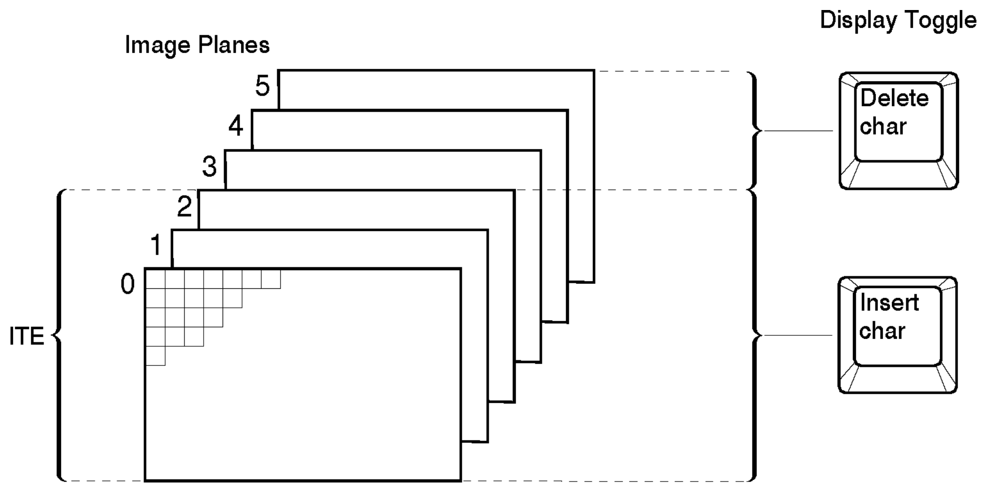

Figure 13-9. ITE on the HP 98549A (C+) Display

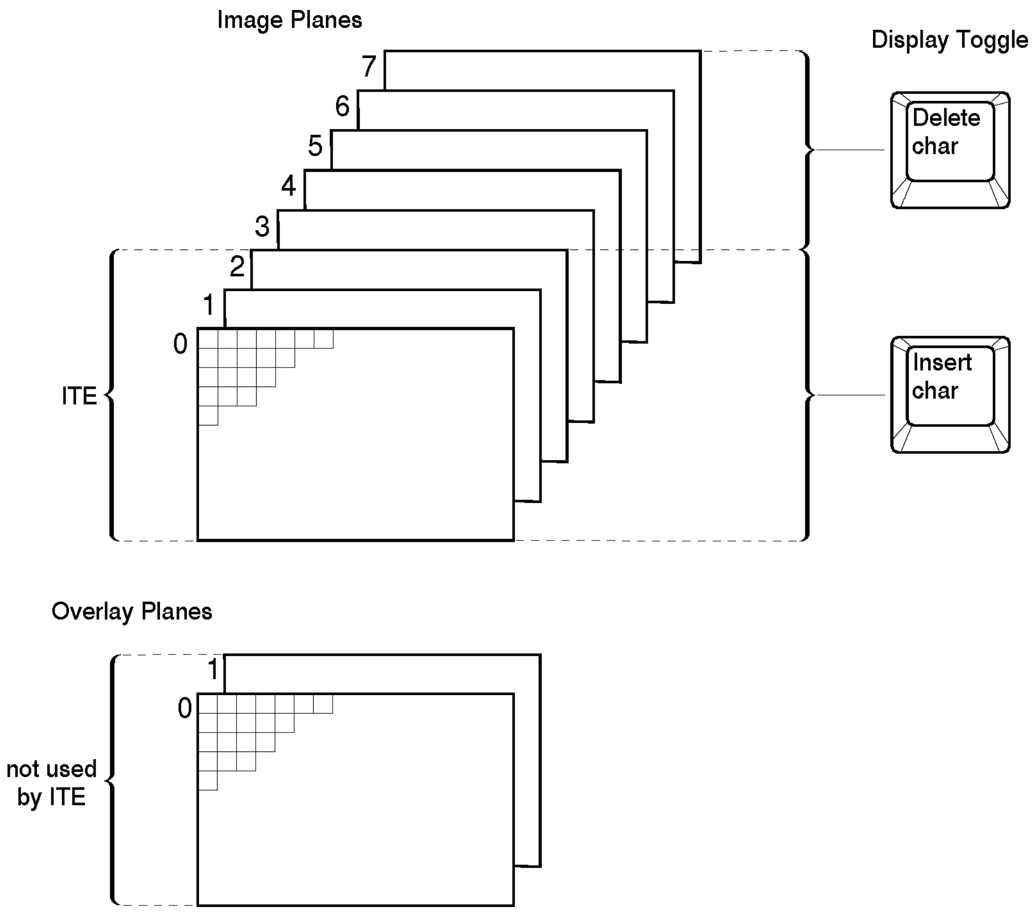

Figure 13-10. ITE on the HP 98550A (CH) Display

The HP 98556A integer accelerator increases the speed of graphics operations. The ITE does not use the accelerator.

For all devices, there may be interactions (i.e. unexpected and/or unpredictable output on the display) if graphics and the ITE are simultaneously accessing the same planes. The ITE and Starbase use a semaphore to mediate access to the frame buffer. The gescapes `SWITCH_SEMAPHORE`, `R_LOCK_DEVICE`, and `R_UNLOCK_DEVICE` enable and disable this semaphore.

Offscreen Memory

The ITE does not use the offscreen frame buffer memory on any of these devices.

On all displays, a hard ITE reset will clear all planes. (This results from pressing the following keys simultaneously: (Shift) (CTRL) (Reset). This hard reset operation will also clear up any bad hardware states that may occur as a result of prematurely aborting a graphics process.

14

The VRX Mono Display

Device Description

The HP A1096A display board fits into a system slot on supported Series 400 SPUs. See the table in the *Introduction* of this manual for supported configurations. The display has a resolution of 1280×1024 pixels with a single frame buffer plane.

Interactions with the ITE

The Internal Terminal Emulator (ITE) operates in the image plane. There may be interactions if graphics and the ITE are active simultaneously in the image plane.

A hard ITE reset (shift-control-reset) will clear the image plane. This hard reset operation will also clear up any bad hardware states that may occur if a graphics process is aborted.

Windows Operation in the Image Plane

The X Window System is supported by the HP A1096A driver. See the *Starbase Graphics Techniques* for a complete discussion of X Window support.

Windows with and without a Backing Store

Consider a graphics window that is partially obscured by another window. What happens when you try to draw graphics to the part of the window that is obscured? There are two options:

- Draw only to the visible parts of the window and ignore any parts that are obscured. An application must receive and handle the exposure events.

- For those parts of the window that are obscured, draw the image to *memory* instead of the frame buffer. When the affected portion of the window is made visible (unobscured), the window system updates the display with the appropriate graphical data from memory. Graphical data stored (retained) in memory is called a **backing store**.

The X Window System supports both options. See the *Starbase Graphics Techniques* for more information on **backing store** in X Windows.

Multiple-Plane Bit/Pixel Support

When `block_read` or `block_write` is used with the `raw` parameter `TRUE`, and `raw` mode is enabled by the `R_BIT_MODE` `gescape`, the driver supports bit/pixel frame buffer access to a single plane.

The `gescape` operation `GR2D_PLANE_MASK` defines a mask that allows reads or writes directly from or to the image plane. It returns the correct data from the visible portions, but not from the obscured portions.

16×16 Fill Pattern

`GR2D_FILL_PATTERN` sets a 16×16 fill pattern used by the driver as a source to fill polygons until the fill pattern is redefined, either by a call to `fill_color` or `fill_color_index` or to the `gescape` `R_DEF_FILL_PAT` or `GR2D_FILL_PATTERN`. The retained raster only supports the `R_DEF_FILL_PAT` 4×4 pattern. The `pattern_define` routine for Starbase is recommended instead of this `gescape`.

Three-Operand Raster Combinations

For a discussion of replacement rules, review the "Drawing Modes" section of the "Frame Buffer Control & Raster Operations" chapter of the *Starbase Graphics Techniques* manual.

Frame Buffer Access

The supported displays provide hardware support for frame buffer access in both plane-major and pixel-major modes. Pixel-major access views the frame buffer as an array of bytes, one byte per pixel, with 1 significant bit per byte. The normal operation of Starbase procedure `block_read` and `block_write` treat the frame buffer in pixel-major mode. Plane-major mode addresses the frame buffer as a single plane, consisting of a packed array of bits, one bit per pixel. This may be used, for example, to transfer data quickly from a data array to a plane. In this driver, plane-major access may be made with the `block_read` and `block_write` procedure by setting the `raw` parameter to `TRUE` and enabling `raw` mode using the `R_BIT_MODE gescape`. (Not all drivers provide this capability.) Details of this form of access are provided below in the section entitled "Starbase Functionality."

The HP A1096A display board produces a resolution of 1280×1024 pixels and provides 1-image plane. There are no overlay planes.

Creation of a special device file is discussed in the next section.

The physical frame buffer is 2048×1024 bytes. The last 768 bytes of each line of the frame buffer (to the "right" of the screen) are not displayed.

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 (right-most) 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 (1023×2048+1279).

For normal `block_read` and `block_write` operations, the data is in the least significant bit of each byte.

Setting Up the Device

Switch Settings

The HP A1096A has two jumper switches. The default switch settings (0,0) configure the display as the internal (console) display.

Table 14-1. Switch Settings Supported on Series 300

Jumper 1	Jumper 2	Select Code
0	0	132
1	0	133
0	1	134
1	1	135

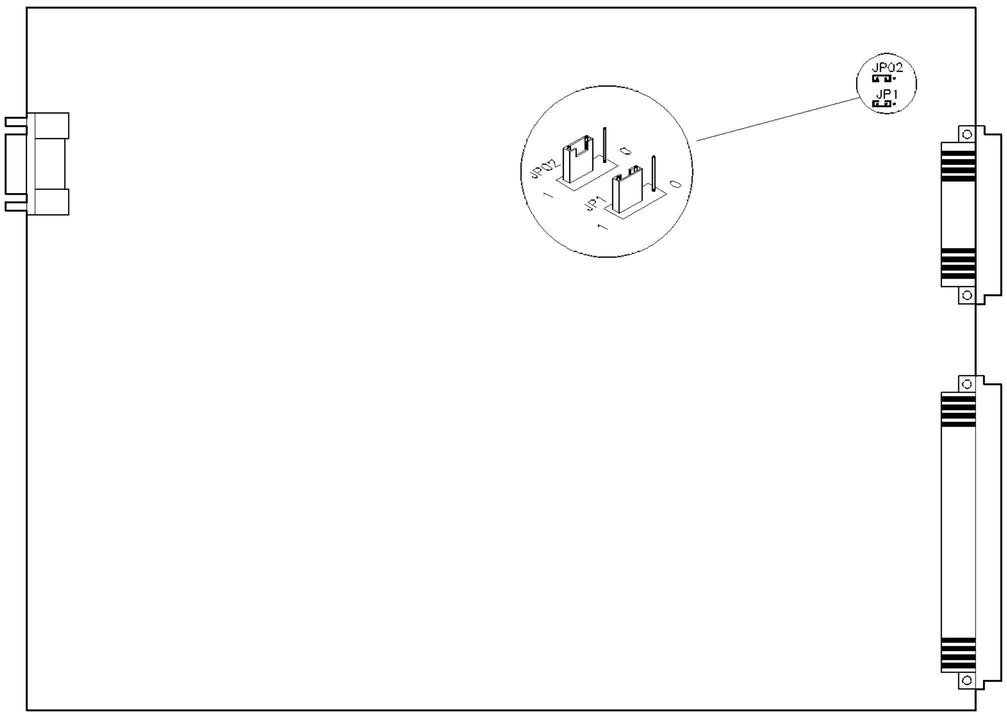

Figure 14-1.

Note The figure shows the `0` and `1` positions for reference, they are not silkscreened on the board.

Special Device Files (`mknod`)

The `mknod` command (see `mknod(8)` in the *HP-UX Reference* manual), creates a special device file that is used to communicate between the computer and the display device. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

The Series 300 `mknod` parameters are:

Character device with a major number equal to `12` and a minor number equal to `0x000000` (internal) or `0x`⟨*sc*⟩`0200` (externally).

Although special device files may be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file, however the name that is suggested for the default device is `crt`.

The following example will create a special device file for the Series 400 internal display. Remember that you must be superuser or root to use the `mknod` command.

```
mknod /dev/crt c 12 0x000000
```

The next example creates a device file for an external configuration:

```
mknod /dev/crt c 12 0x⟨sc⟩0200
```

Linking the Driver

Shared Libraries

The shared HP A1096A Device Driver is the file named `libdda1096.sl` in the `/usr/lib` directory. The device driver will be explicitly loaded at run time by compiling and linking with the starbase shared library `/usr/lib/libsb.sl`, or by using the `-l` option `-lsb`.

Examples

To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11 -L/usr/lib/X11R5\
-lXwindow -lsb -lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

Archive Libraries

The HP A1096A Device Driver is located in the `/usr/lib` directory with the file name `libdda1096.a`. This device driver may be linked to a program using the absolute path name `/usr/lib/libdda1096.a` or an appropriate relative path name, or by using the `-l` option `-ldda1096`.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libdda1096.a`
2. an appropriate relative path name
3. the `-ldda1096` option with the `LDOPTS` environmental variable exported and set to `-a archive`.

By default, the linker program `ld(1)` looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using **ksh**(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldda1096 -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldda1096 -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldda1096 -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Initialization

Parameters for gopen

The `gopen` procedure has four parameters: Path, Kind, Driver, and Mode.

Path This is the name of the special device file created by the `mknod` command as specified in the last section, e.g. `/dev/crt`.

Kind This parameter must be `OUTDEV`, unless used for a graphics window, in which case `OUTINDEV` may be used.

Driver The character representation of the driver type. This is `hpa1096` modified to meet the syntax of the programming language used, namely:

`"hpa1096"`	*for C.*
`'hpa1096'//char(0)`	*for FORTRAN77.*
`'hpa1096'`	*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 *Starbase Graphics Techniques* for a description of `gopen` actions when accessing an X Window.

`SPOOLED`	Raster devices cannot be spooled.
`MODEL_XFORM`	Shading is not supported for this device. However, opening in `MODEL_XFORM` mode will affect how matrix stack and transformation routines are performed.
`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. Frame buffer is cleared to 0s. 2. The display is enabled for reading and writing.
`RESET_DEVICE`	Open and initialize the device as follows:

1. The hardware state is reinitialized to its boot-up state.
2. Frame buffer is cleared to 0s.
3. The display is enabled for reading and writing.

Note `SPOOLED` and `MODEL_XFORM` flag bits have no device dependent effects.

Syntax Examples

To open and initialize an HP A1096A device for output:

C programs:

```
fildes = gopen("/dev/crt",OUTDEV,"hpa1096",INIT);
```

FORTRAN77 programs:

```
fildes = gopen('/dev/crt'//char(0),OUTDEV,'hpa1096'//char(0),INIT)
```

Pascal programs:

```
fildes := gopen('/dev/crt',OUTDEV,'hpa1096',INIT);
```

Special Device Characteristics

Device Coordinate Addressing

For device coordinate operations, location (0, 0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is (1279, 1023).

Starbase Functionality

Commands Not Supported

The following commands are not supported. If one of these commands is used by mistake, it will be ignored and not cause an error.

```
alpha_transparency            hidden_surface
backface_control              light_ambient
bank_switch                   light_attenuation
bf_alpha_transparency         light_model
bf_control                    light_source
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              shade_range
bf_texture_index              surface_coefficients
contour_enable                surface_model
define_color_table            texture_index
define_contour_table          texture_viewport
define_texture                texture_window
define_trimming_curve         viewpoint
deformation_mode              zbuffer_switch
depth_cue
depth_cue_color
depth_cue_range
```

Conditionally Supported Procedures

The following procedures are supported under the listed conditions:

`block_read`, `block_write`	When the `raw` parameter is set to `TRUE`, data is arranged in the form of 8 pixels per byte (bit = pixel). Reads and writes must be performed on byte or word boundaries. Clipping is not performed in `raw` mode.

`with_data`	`partial_polygon_with_data3d`
	`polygon_with_data3d`
	`polyhedron_with_data`
	`polyline_with_data3d`
	`polymarker_with_data3d`
	`quadrilateral_mesh_with_data`
	`triangle_strip_with-data`
	
	Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

Fast Alpha and Font Manager Functionality

This device driver supports raster text calls from the fast alpha and font manager libraries. See the *Fast Alpha/Font Manager Programmer's Manual* for further information.

Parameters for gescape

The `hpa1096` driver supports the following `gescapes`. Detailed information about these functions can be found in Appendix A.

- `GR2D_FILL_PATTERN`—Define 16×16 dither and fill pattern.
- `GR2D_PLANE_MASK`—Overrides the mask.
- `R_BIT_MODE`—Bit mode.
- `R_DEF_FILL_PAT`—Define fill pattern.
- `R_FULL_FRAME_BUFFER`—Full frame buffer.
- `R_GET_FRAME_BUFFER`—Read frame buffer address.
- `R_LOCK_DEVICE`—Lock device.
- `R_UNLOCK_DEVICE`—unlock device
- `SWITCH_SEMAPHORE`—semaphore control

Performance Tips

- Certain two-operand drawing modes may be done faster than others. The absolute modes (`ZERO` (0) and `ONE` (15)) are the fastest. Rules dependent only on one operand (e.g., ⟨*source*⟩ (3) or `not` ⟨*dest*⟩ (10)) are somewhat slower. Rules dependent on both operands (e.g., `xor` (6)) are the slowest.
- Buffering of graphics operations is done in this driver to enhance performance. If `buffer_mode` is turned off or many calls are made to `make_picture_current` then performance may decrease.
- Performance optimizations have been made so that sequential calls of the same output primitive with no intervening attribute change or call to a different primitive are processed faster. For example, the sequence `polygon, polygon, polyline, polyline` is faster than `polygon, polyline, polygon, polyline`. The `line_color, polyline, polyline` calls are faster than `line_color, polyline, line_color, polyline`.
- For the best performance when using bit/pixel block write (`raw` mode `TRUE`, `R_BIT_MODE` enabled), the following conditions must be met:
  1. Source rows should be an even number of whole bytes (that is, `dx` should be a multiple of 16).
  2. Destination rows should be aligned on 8-pixel boundaries (that is, `x` should be a multiple of 8).
  3. Source rows should be aligned.

Note When drawing in a graphics window, drawing and filling performance will be significantly lower if the window raster extends more than 1024 device coordinates outside the screen in any direction, either because of its size or its current position on the screen. There is a significant additional performance cost associated with drawing to a retained rather than an unretained raster.

15

The HP 9836A Device Driver

Device Description

This device driver is used to provide graphics output on the Series 300 HP 98546A Display Card. This display is a bit-mapped device which has a resolution of 512×390 pixels with a single plane of frame buffer. A separate non-bit mapped alpha plane allows text and graphics to be manipulated independently.

This display card provides compatibility with programs written for HP 9836A Series 200 models and for HP 98204B displays.

The interface for this device plugs into an I/O slot of supported SPUs. See table 1-8 in the introduction of this manual for the SPUs which support this device.

The display is organized as an array of bytes, with each byte representing 8 pixels on the display. Typically, the user does not need to directly read or write pixels in the frame buffer. However, for those applications which require direct access, Starbase provides the `gescape R_GET_FRAME_BUFFER`, which returns the virtual memory address of the beginning of the frame buffer (this `gescape` is discussed in the appendix).

Frame buffer locations are addressed relative to the returned address. The first byte of the frame buffer (byte 0) represents eight pixels in the upper left corner of the screen. The Most Significant Bit (MSB) of that byte holds the pixel in the upper left corner of the display. The Least Significant Bit (LSB) of the first byte holds the right-most pixel in the byte (that is, pixel number 8). Byte 1 is immediately to its right. Byte 63 holds the last 8 pixels on the top line. Byte 64 hold the 8 pixels below the first line. The last (lower right corner) set of 8 pixels on the screen is in byte number 24,959 (389×64+63).

This display does not support the Windows/9000 nor the X Windows system.

Setting Up the Device

Switch Settings

For normal operation of this device, there are no switches to set.

Special Device Files (mknod)

The `mknod` command (see `mknod` in the man pages), creates a special device file which is used to communicate between the computer and the peripheral device. The name of this special device file is passed to Starbase in the `gopen` procedure. Since superuser capabilities are needed to create special device files, they are normally created by the system administrator.

The `mknod` parameters are the character device with a major number of 12 and a minor number of 0. Although special device files can be made in any directory of the HP-UX file system, the convention is to create them in the `/dev` directory. Any name may be used for the special device file; however, the name that is suggested for these devices is `crt`. The following example will create a special device file for this device. Remember that you must be superuser or root to use the `mknod` command.

```
mknod /dev/crt  c  12  0x000000
```

Linking the Driver

The HP 9836A Device Driver is located in the `/usr/lib` directory with the file name `libdd9836a.a`. This device driver may be linked to a program using the absolute path name `/usr/lib/libdd9836a.a`, an appropriate relative path name, or the `-l` option `-ldd9836a`. To compile and link a C program for use with the shared library driver, use:

```
cc example.c -I/usr/include/X11R5/x11\
-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

or with FORTRAN use,

```
F77 example.f -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm  -o example
```

or with Pascal use,

```
pc example.p  -Wl,-L/usr/lib/X11R5 -lXwindow -lsb\
-lXhp11 -lX11 -ldld -lm -o example
```

For details, see the discussion of the `gopen` procedure in the section *To Open and Initialize the Device* in this chapter.

For Archive Libraries

The archive device driver is located in the `/usr/lib` directory with the file name `libdd9836.a`.

You can link this device driver to a program by using any one of the following:

1. the absolute path name `/usr/lib/libdd9836.a`
2. an appropriate relative path name
3. the `-ldd9836` option with the `LDOPTS` environmental variable exported and set to `"-a archive"`.

By default, the linker program `ld`(1) looks for a shared library driver first and then the archive library driver if a shared library was not found. By exporting the `LDOPTS` variable, the `-l` option will refer only to archive drivers.

Examples

Assuming you are using `ksh`(1), to compile and link a C program for use with this driver, use:

```
export LDOPTS="-a archive"
```

and then:

```
cc example.c -ldd9836 -L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -lm  -o example
```

or for FORTRAN, use:

```
F77 example.f -ldd9836 -Wl,-L/usr/lib/X11R5 -lXwindow\
 -lsb1 -lsb2 -lXhp11 -lX11 -o example
```

or for Pascal, use:

```
pc example.p -ldd9836 -Wl,-L/usr/lib/X11R5 -lXwindow\
-lsb1 -lsb2 -lXhp11 -lX11 -o example
```

Initialization

Parameters for gopen

The `gopen` procedure has four parameters: Path, Kind, Driver, and Mode.

Path	The name of the special device file created by the `mknod` command as specified in the last section, e.g. `/dev/crt`.
Kind	Indicates the I/O characteristics of the device. This parameter must be `OUTDEV` for this driver.
Driver	The character representation of the driver type. This is `hp9836a`, `hp98546a`, or `hp98204b` modified to meet the syntax of the programming language used, namely:

`"hp9836a"`	*for C.*
`'hp9836a'//char(0)`	*for Fortran77.*
`'hp9836a'`	*for Pascal.*

Mode — The mode control word consisting of several flag bits which are *or* ed together. Listed below are those those flag bits which have device-dependent actions. `SPOOLED` flag bits have no affect for this driver. Those flags not discussed below operate as defined by the `gopen` procedure.

- `SPOOLED`—cannot spool raster devices.
- `0`—open the device and initialize the software color map
- `INIT`—open and initialize the device as follows:
  1. Frame buffer is cleared to 0s.
  2. The color map is set to its default values.
  3. The display is enabled for reading and writing.

Syntax Examples

To open and initialize an HP 9836A device for output:

C Syntax Examples

```
fildes = gopen("/dev/crt",OUTDEV,"hp9836a",INIT);
```

FORTRAN77 Syntax Examples

```
fildes = gopen('/dev/crt'//char(0), OUTDEV,'hp9836a'//char(0),INIT)
```

Pascal Syntax Examples

```
   fildes = gopen('/dev/crt',OUTDEV,'hp9836a',INIT);
```

Special Device Characteristics

For device coordinate operations, location (0, 0) is the upper-left corner of the screen with X-axis values increasing to the right and Y-axis values increasing down. The lower-right corner of the display is therefore (511, 389).

Starbase Functionality

Exceptions to Standard Starbase Support

The following commands are supported under the listed conditions:

`await_retrace`	This routine has no effect on this display.
`block_read, block_write`	When the `raw` parameter is set to `TRUE`, it indicates that `DATA` is arranged with 8 pixels/byte. The data rounds to a pixel in the X-axis direction that aligns with a byte or word boundary. Clipping of the data is not performed in `raw` mode.
`define_color_table`	Since there is no hardware color map, this command defines a software color map on black and white devices.
`inquire_color_table`	This command returns the software color map values on black and white devices.
`interior_style`	Only the `INT_SOLID`, `INT_HATCH`, and `INT_HOLLOW` styles are supported.
`text_precision`	Only `STROKE_TEXT` precision is supported.

`with_data`	`partial_polygon_with_data3d`
	`polygon_with_data3d`
	`polyhedron_with_data`
	`polyline_with_data3d`
	`polymarker_with_data3d`
	`quadrilateral_mesh_with_data`
	`triangle_strip_with-data`
	Additional data per vertex will be ignored if not supported by this device. For example, contouring data will be ignored if the device does not support it.

Commands Not Supported

The following commands are not supported. If one of these commands is used by mistake, it *will not* cause an error.

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

Index