Standard Text Interface
For Graphics Devices

Revision 8.13
March 1, 2000

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information that is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright © 1983-1999 by HEWLETT-PACKARD COMPANY All Rights Reserved

Table of Contents

Table of Figures

1 Purpose

The purpose of this document is to define and explain the format and functionality of the STI ROM in enough detail to allow STI developers and those who need to interface to graphics devices to make their contributions efficiently and correctly.

The purpose of the STI itself is to provide a means whereby the SPU Boot ROM and Operating System can perform the graphics device dependent text operations and self test without having specific device dependent information. Instead, the device dependent information is contained in code supplied in the STI ROM. A much more detailed set of objectives is listed below.

Note that the first products using STI were released in 1990 and the original STI specification was also released in 1990. The last revision to the STI specification was made in 1991. There have been numerous additions to the STI capability since that time and this revision attempts to bring the STI specification up to date.

> *Editorial comments have been added, and are shown in italic, where there is a difference in the original specification and this revision. This is mostly the case for some of the comments that have been dated by new hardware additions.*

1.1 Usage

STI has three primary modes of usage:

1. Boot console support. The Boot ROM (SPU system firmware) uses STI in a device independent manner to display boot time messages. In PC type boxes this type of functionality has been provided by the combination of VGA hardware and the VGA Bios.

2. OS console support. The Operating System also uses STI in a device independent manner to display system messages. When a graphics card is used to display console messages, STI is the mechanism that is used.

3. OS runtime support. The Operating System also uses STI for runtime support to provide services to users of the graphics device, as well as its own use. A primary example of users of the graphics device is the X11 server and an example of the Operating System runtime use is interrupt support.

1.2 Assumptions

Several assumptions have been made about what type of devices will use the Standard Text Interface.

These assumptions include:

- All devices will be bitmapped graphics devices. This implies that none of the devices necessarily have separated, dedicated alpha and graphics planes although some planes may tend to be used primarily for text.

  Note that other, non-graphics devices, have used the STI concept. It turns out that STI is useful in the development of new hardware. It has been used many times to quickly add devices for prototyping purposes.

- Off screen memory will not necessarily be available.

- If offscreen display is available, it will be to the right and/or below the onscreen display.

- The device will not necessarily support hardware block moves although the STI ROM must still provide a block move routine, even if it requires the use of sequential reads and writes to do the block move.

- The frame buffer configuration can vary significantly from device to device.

- Color may or may not be supported.

- Only merged 680X0 and PA-RISC SPU platforms will be supported.

  Note that Motorola based HP Workstations production ended many years ago. The original specification covered both the 68000 family and the PA-RISC family of HP Workstations.

- HP-UX and Domain, and possibly OSF Operating Systems will be supported.

  Note that many other Operating Systems have been ported to use STI. HP-UX, which is the Hewlett-Packard implementation of UNIX was and still is the primary user of STI. Domain was the Operating System used by the Apollo brand computer that Hewlett-Packard acquired in the early 1990s. STI has also been used by HP-RT, the HP Real Time OS. Other Operating Systems that have used STI are FreeBSD, NextStep and Mach, and there is currently a port of Linux to PA-RISC that is using STI.

- No microcode will be used by the STI routines to support the necessary functionality.

The intent was not to limit STI usage of microcode, but to not require a user of STI to manage microcode supplied by STI. STI has been used with hardware that required microcode, but STI controlled the microcode download to the hardware, transparent to the user of the STI routines.

1.3 Objectives

Musts:

- Provide a mechanism whereby the SPU can initialize the graphics device to a known state and determine device configurable characteristics (e.g. frame buffer size).

- Provide an interfacing mechanism for all SGC cards, which will work without change in any SPU that supports SGC. (PA-RISC and 68000).

  The original bus in the PA-RISC and 68000 workstation products was SGC but STI has since been used with many hardware and bus architectures.

- Insure that the overall system cost that the interfacing mechanism generates is minimized.

- Provide a mechanism that will continue to be usable for products that will be developed in the next 2-3 years.

  STI has been in use for over 10 years now. The architecture provided by STI turned out to be highly extensible and very flexible.

- Insure that the schedules of SGC products currently under development are not adversely affected by the solution.

  We were developing the STI architecture simultaneously with the first version of PA-RISC workstation hardware. Obviously we wanted to make sure that we did not impact product delivery schedules.

- Provide a mechanism whereby the SPU can easily determine what type of graphics device, if any is in each slot.

- Provide a mechanism whereby the SPU Boot ROM can use the device as system console and communicate with the user via text output to the graphics device.

- Provide an interfacing mechanism such that the existing releases of the SPU Boot ROM can perform the necessary text operations to new graphics hardware without any changes (after initial STI release) to the Boot ROM. (De-couple STI/Boot ROM releases).

This was the primary motivation behind STI. Prior to the invention of STI, new graphics that were developed for both new and existing workstations and servers required System Firmware as well as System Kernel changes to support the new graphics hardware. This was a real problem with syncing up the system software and graphics hardware releases.

- Provide a mechanism whereby the SPU Kernel/ITE is able to perform necessary text operations to the graphics device.

- Provide an interfacing mechanism such that the existing releases of the Kernel/ITE can perform the necessary text operations to new graphics hardware without any changes (after initial STI release) to the Kernel/ITE. (De-couple HW/SW releases).

This was the primary motivation behind STI. Prior to the invention of STI, new graphics that were developed for both new and existing workstations and servers required System Firmware as well as System Kernel changes to support the new graphics hardware. This was a real problem with syncing up the system software and graphics hardware releases.

- Insure that all system configurations have available at least one font to support the necessary text operations.

- Provide a mechanism whereby graphics product designers can implement test functionality that will provide the SPU with pass/fail information at the FRU (Field Replaceable Unit) level.

High Wants:

- Minimize the amount of SPU ROM and RAM that the Boot ROM requires in order to use the graphics device as system console.

- Minimize the level of ongoing support needed by the kernel group to provide ongoing support for the ITE.

- Provide a standard text interfacing mechanism for use by other graphics devices in addition to SGC graphics device.

We knew that SGC would be just the first bus implementing graphics and we wanted to make sure that there were no bus specific limitations.

1.4 Plans for achieving objectives

Note that all the comments in this section were retained from the original specification for historical purposes only.

It is believed that the following strategy allows us to best meet the objectives listed above. The strategy focuses on defining and implementing device independent interfaces and minimizing system cost.

Determine which functionality is necessary for the Boot ROM and kernel to use the graphics device.

Of this functionality, determine which is dependent on the characteristics of a particular graphics device and provide it in an STI ROM in the graphics hardware.

Implement the device independent functionality in the SPU such that systems with multiple SGC devices will only incur the cost of such functionality once.

Provide a simple, well defined, device independent interface between the SPU and the graphics device.

Encourage the SPU designers to provide a simple, well defined interface between the ITE and the Boot ROM to minimize duplication of functionality in the SPU.

2 Definitions

2.1 Graphics

Graphics hardware is commonly broken into frame buffer space, which is the memory of the pixels that are displayed on the screen, and control space, which are registers to control the card. STI also creates a space called the privileged space.

2.1.1 Frame Buffer

The frame buffer is where the pixels that are shown on the screen are located.

2.1.2 Control Space

Control space provides for access to registers that control the video card. Examples include video timing registers, etc.

2.1.3 Privileged Space

The privileged space normally contains control registers that you would not want any user program to touch. One such example would be the reset bit that resets the card. If this was accessible through user space then any user program could reset the card. This could potentially lead to panic of the kernel.

2.2 Systems

Systems consist of the SPU, Boot ROM and the Kernel.

2.2.1 SPU

SPU is the system processing unit. It is an all-encompassing method of describing the box that includes CPU, memory and all other pieces that are normally thought of as the computer system.

2.2.2 Boot ROM

The Boot ROM is the firmware associated with the CPU motherboard that controls all the base system hardware.

2.2.3 Kernel

The Kernel is the core part of an Operating System.

2.3 Bus

Bus defines the connection mechanism between the system and the graphics card.

2.3.1 SGC

The Standard Graphics Connection bus, abbreviated SGC, is defined in a separate document A-5960-1585-1. It was defined as a standard interface to connect graphics products to HP workstation hosts.

There were multiple form factors available.

2.3.2 GSC

The GSC bus was originally known as the Graphics System Connect and later came to be called the Gonzo System Connect, and is defined in a separate document. There were multiple versions including the original GSC, GSC-1X, GSC+, GSC-1.5X and GSC-2X.

The GSC bus is a synchronous, 32 bit I/O bus used in many of Hewlett-Packard's computers. GSC features a multiplexed address/data bus, a parity bit, basic protocol with extensions to meet both workstation and server needs and a data rate up to 256Mbyte/sec.

There were multiple form factors available.

The GSC 1.5X extensions define write coalescing. The usage example would be writes to a graphics device. The processor writes to the graphics FIFO/frame buffer by executing single word or double word store instructions to an I/O space address. Writes are issued to consecutive memory locations, hardware is added that evaluates the addresses and coalesce contiguous transfers into transactions with larger data payloads.

The GSC 2X extensions address the frequency at which data is driven onto the physical bus. The 2X extensions define a protocol whereby the device driving data can switch the data on every rising and falling edge of the GCLK.

2.3.3 PCI

The PCI bus is defined by PCI SIG and is currently at revision 2.2 (December 18,1998).

2.3.4 Other

STI is adaptable to other buses and has been used successfully on VME and others, including proprietary non-HP system buses.

2.4 ENDIAN

To be consistent with the SGC specification, the "Motorola" endian interface is used. This means that it has big endian byte and halfword (16 bit) addressing but the bits are numbered little endian. The bus does not support bit addressing and so this is just a matter of convention. Byte transactions are supported and are always aligned to the proper location within the 32-bit word.

2.4.1 Big Endian

Most Significant		Least Significant	
Byte 0	Byte 1	Byte 2	Byte 3
31 ------ 24	23 ------ 16	15 ------ 8	7 ------ 0
Halfword 0		Halfword 1	

Figure 1 – Big Endian

2.4.2 Little Endian

Least Significant		Most Significant	
Byte 3	Byte 2	Byte 1	Byte 0
31 ------ 24	23 ------ 16	15 ------ 8	7 ------ 0
Halfword 1		Halfword 0	

Figure 2 – Little Endian

2.5 Nomenclature

A "0x" in front of a value indicates the value is a hexadecimal representation (unsigned unless otherwise noted). All other values are assumed to be decimal.

3 Overview

3.1 STI ROM Structure

The general structure of the ROM can be broken down into the following sections:

1. Device Data
2. Routines
3. Font Data
4. Other Data

The Device Data (DD) section includes the graphics ID, other information and pointers to other functions and sections within the ROM.

The Routines section includes the following routines:

- Initialization Routine (INIT_GRAPH)

  A routine to put the graphics system into an initialized state that will support text output to the device. It will do such things as enable the needed frame buffer planes, setup the color map and return configuration information that can vary from graphics device to graphics device.

- State Management Routine (STATE_MGMT)

  This routine provides the capability to save the current state of the graphics device before it is initialized and then restore this state at some later time.

- Font Unpack/Move (FONT_UNP/MV)

  A routine to unpack a font character out of the font storage space and put it at a (x, y) location in the frame buffer.

- Block Move Routine (BLOCK_MOVE)

  A routine to move a variable size block of data from one (x, y) pixel location in the frame buffer to another. This will be used to move data within the frame buffer. This routine can also be used to clear a variable size block of data within the frame buffer.

- Self-Test Routine (SELF_TEST)

  A routine for testing the device at power-up.

- Exception Handling Routine (EXCEP_HDLR)

  A routine for handling interrupts and bus errors.

- Inquire Configuration Routine (INQ_CONF)

  A routine for providing information about the current configuration of the device. This routine is primarily used in support of the GCDESCRIBE functionality needed by the HP-UX kernel.

- Set Color Map Entry (SET_CM_ENTRY)

  A routine that allows control of color map entries that is not directly used by STI. This was added to support the "early console" capability.

- DMA Control (DMA_CONTROL)

  A routine for setting up and managing DMA activity to or from the STI based device.

- Flow Control (FLOW_CONTROL)

  A routine for managing the IO transactions to the device. Normally this is for maintaining the high-water and low-water marks associated with buffered devices.

- User Timing (USER_TIMING)

  A routine for allowing user based code to request a reconfiguration of the monitor timing.

- Process Management (PROCESS_MGR)

  A routine to handle unique HP-UX kernel requirements normally used to control implicit locking devices.

- STI Utility (STI_UTIL)

  A routine that provides generic capabilities to user based processes. This is for additional functionality that is not controlled by one of the other STI routines.

The Font Storage area includes all of the appropriate fonts necessary for the proper operation of the device.

The Other area includes items like:

- List of device Regions to map into memory.
- Monitor timing information
- User unique data

3.2 ROM Format

In several places in the STI ROM, addresses and/or offsets are stored. These addresses and offsets are all, unless otherwise noted, relative to the start of the STI ROM (which is, almost always, the start of the device's I/O space). These addresses and offset must have the following format unless otherwise noted. They must all be values that contain a count of the number of bytes from the beginning of the STI ROM to the address of interest. In addition, the count must point to the byte that contains the valid data. For example, if we wanted to add a value in the ROM which is the offset to the "Local ROM rev", then this offset would contain the value 0x0f.

> *Note that the comment about the start of the device's I/O space is not true for PCI based devices. Also the example offset given is for the Byte Mode ROM, which is described later.*

Pointers to the various places within the ROM can then be created by adding these offsets to the address of the start of STI ROM space. (See the section on Device Region List for more information on this). These pointers can then be used as byte, half-word or word pointer by the routines as they choose. The users of these pointers must be aware however that using them as half-word or word pointers will probably cause the SPU to do two cycles for the access since the pointer is not aligned to a word or half-word. The user may want to adjust their value for the proper alignment in such cases. In addition, some machines may have harsher restrictions on the type of pointers allowable and their alignment. STI code developers should check with 68000 and PA-RISC experts to get further details.

For consistency, all unused locations must contain zeros. This includes unused bit fields within locations that contain flags. In addition, if a location is defined to contain a pointer to a routine but that routine is not available, the pointer must be set to zero to indicate that the routine does not exist. Hardware designers must not map readable registers on top of unused locations in the ROM as even the unused locations are considered when the CRC is done. Registers can still be mapped over the upper three unused bytes of each location.

> *Again note that the comment about mapping registers over the upper three unused byte locations pertains to Byte Mode ROM only.*

3.3 Usage considerations

3.3.1 Byte Mode ROM in low address

The STI ROM must reside in the lowest address locations of the graphics device and must contain an even number of bytes.

The graphics device is only allowed to map its ROM onto the least significant byte of each word in its address space. Previous ROM definitions have allowed mappings to bytes, half-words or words. Even with these options most devices have chosen to use only the least significant byte due to the simplification it offers in hardware design.

Because of this mapping scheme, the code provided in the STI ROM is not designed for execution directly from ROM. Rather, it must be loaded into system RAM in a packed format (every fourth byte in ROM becomes contiguous in RAM) and then executed.

3.3.2 PCI Expansion ROM Support

PCI based graphics devices can use the PCI Expansion ROM space in addition to the memory mapped I/O space, but they must at least support the PCI Expansion ROM space.

3.3.3 PCI Multiple Bar Support

The graphics device, on PCI, can support multiple Bars

3.3.4 PCI Speed and Size

All variants of the PCI bus are supported. Currently HP does not have any AGP enabled workstations. Our current workstations support both the 32 and 64 bit PCI address ranges as well as both the 33 and 66 MHz speed selections.

3.3.5 SGC

SGC was only a single speed bus.

3.3.6 GSC

GSC has been available at basically two speeds, either 30Mhz or 40Mhz. See the GSC spec for details.

3.3.7 HP-UX 32 and 64 bit OS

Current STI supports both the 32 and 64 bit versions of the HP-UX Operating System.

4 ROM

4.1 Byte, Word and PCI formats

In PA-RISC systems based on SGC or GSC IO bus there are 4 dedicated spaces for graphics (each of which is 32Mbytes). These spaces are 0xf4000000, 0xf6000000, 0xf8000000 and 0xfa000000. The first two 32Mbyte spaces can be combined into a single 64Mbyte space, and likewise, the second two 32Mbyte spaces can be combined into a single 64Mbyte space. Note that the 2nd and 3rd 32Mbyte spaces cannot be combined into a single 64Mbyte space. So in these systems you can have up to 4 separate graphics cards with each requiring a maximum of 32Mbytes or some combination less than 4 cards if one or more require 64Mbytes. If a graphics card does not require the full range (32Mbytes or 64Mbytes) the remaining space is not available for other IO.

For SGC and GSC bus based systems, the STI ROM is architected to live at the beginning of the hardware address space. For example, if you have a graphics card that only requires 32Mbytes and is in a slot that maps the based address of the card to 0xf4000000 then the ROM would start at 0xf4000000. There are no architectural requirements for the size of the ROM. Typically you will find that the ROM ranges in size from 64Kbytes on older hardware to 256Kbytes on newer graphics hardware.

Note that there is a single exception to the rule that the STI ROM lives at the base of the graphics hardware (for SGC and GSC based graphics). The 712/60-80-100, the 715/100 and the 725/100 products had built in graphics (on the GSC bus) that did not have a physical ROM for the graphics firmware. The graphics firmware (STI ROM) was included as part of the base System ROM and the Boot ROM and Kernel were special cased to handle this one exception. All other SGC and GSC based graphics cards have followed the standard with the ROM at the base of the graphics hardware.

Some of the early SPU based on PCI still had the requirement that the STI ROM live at the base HPA of the graphics hardware. Newer systems do not require this because the BOOT ROM and HP-UX kernel have been changed to use the PCI expansion ROM mapping mechanism associated with PCI based systems.

Prior to PCI based systems, STI images were available in two primary formats referred to as BYTE MODE and WORD MODE. The distinction is how the ROM is mapped into the address space.

With the adoption of the PCI Local Bus architecture, a modification to the WORD MODE format was made.

4.1.1 BYTE MODE

Byte mode ROMS will be mapped such that the least significant byte in PA-RISC address space is all that is valid. The top three bytes, in a long word read, will not be used. Obviously, you can not execute code out of the BYTE MODE ROM.

The following figure shows how the least significant byte is mapped into PA-RISC memory mapped IO space. The area in gray depicts the valid data for each long word address.

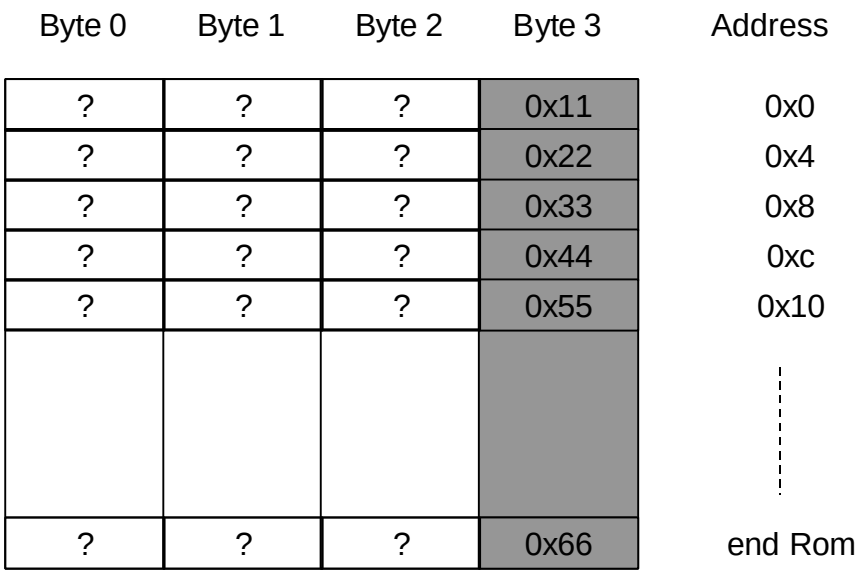

Figure 3 - Byte Mode ROM

Note that in the figure if you perform a long word read of address 0x0 you would get 0x??????11 where the ?? is indeterminate data because only the least significant byte is valid.

For some hosts, the low byte will be duplicated to the other 3 byte positions and for other hosts it is whatever the bus termination provides. Still yet other hosts might drive out whatever was last latched in the bus transceivers.

4.1.2 WORD MODE

Word mode ROMS map all four bytes into the PA-RISC address space so that a user does not have to read 4 bytes and pack them into a word in order to execute a single

instruction. Based on this you could actually execute out of the ROM, but it is highly recommended that this not be done as some graphics cards have used self modifying code for operation.

The following figure shows how all four bytes are mapped into PA-RISC memory mapped IO space. The area in gray depicts the valid data for each long word address.

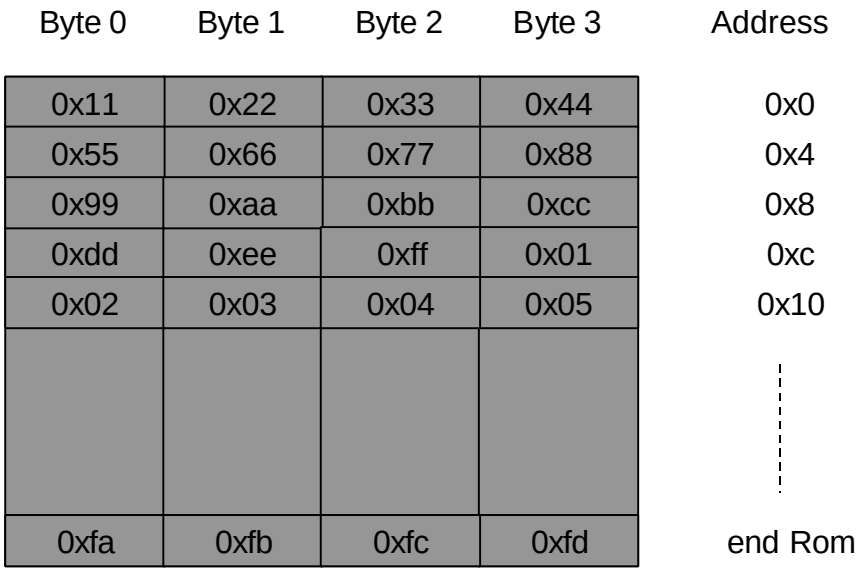

Figure 4 - Word Mode ROM

Note that in the figure if you perform a long word read of address 0x0 you would get 0x11223344 because all four bytes are mapped.

4.1.3 PCI

The PCI Local Bus Architecture defines an Expansion ROM. It describes the header format to be used in the Expansion ROM images on the PCI cards.

4.1.3.1 Multiple ROM Images in a PCI ROM

The PCI architecture allows for multiple ROM images in a single physical ROM as shown in the following figure. In addition, there is nothing to prohibit code supporting different architectures (x86 and PA-RISC) or supporting different operating systems (DOS and HP-UX).

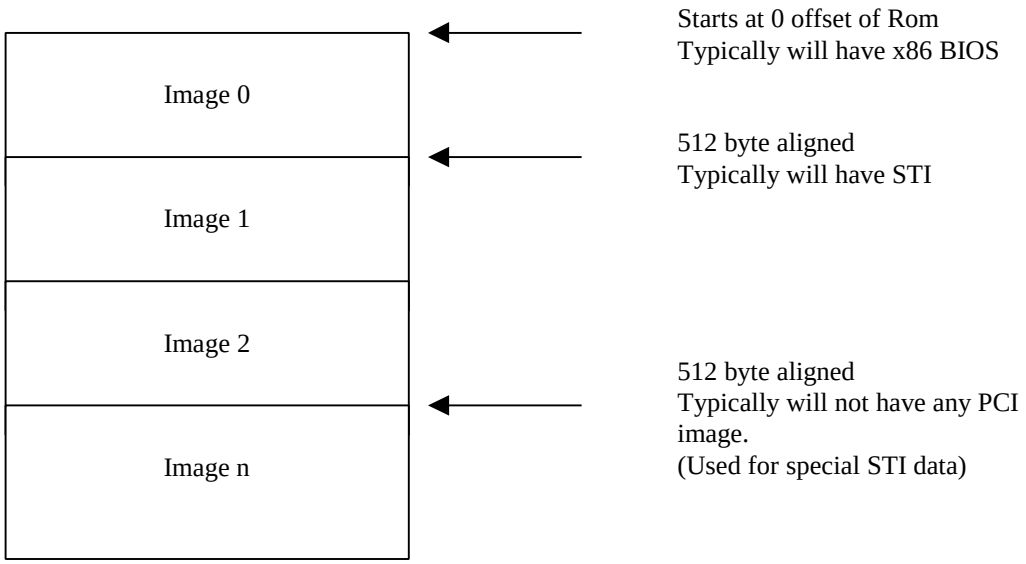

Figure 5 - PCI ROM Image

4.1.3.2 Details of a single ROM image

This document concerns STI and not x86 BIOS so we will focus on the details of what an STI image in a PCI ROM would look like. In the figure below you can see the basic organization of the parts of PCI Image and how they relate to STI. The normal WORD MODE STI image is not changed, it is just appended to the PCI ROM Header and the PCI data structure.

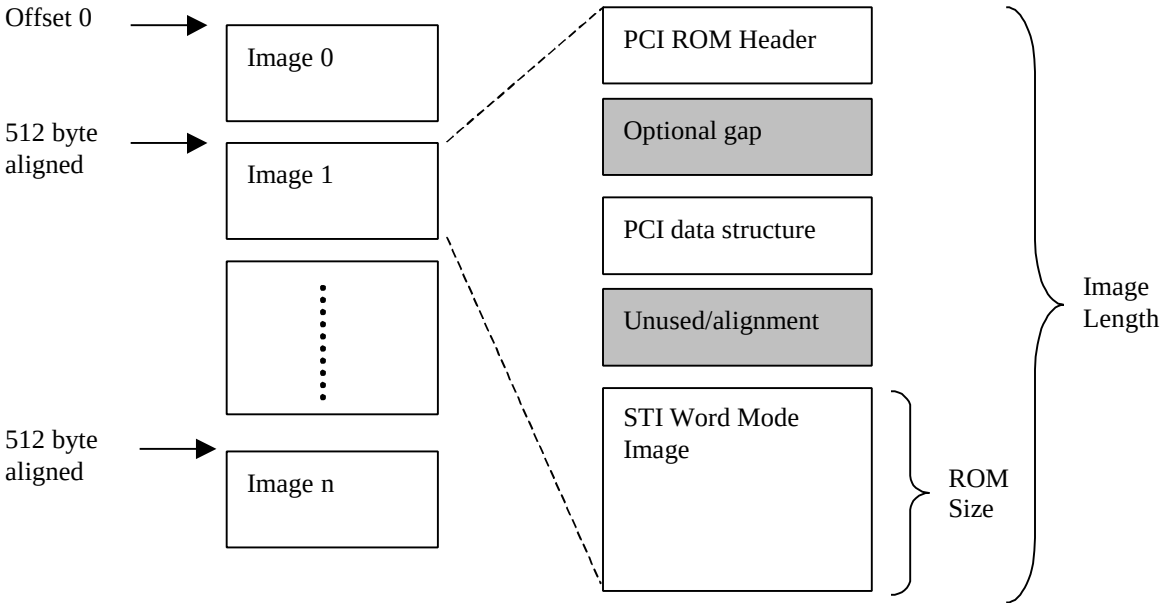

Figure 6 - STI Image in PCI Rom

4.1.3.3 PCI ROM Header Details

Note that all offsets are offsets from the beginning of the PCI ROM Header to the first valid byte of the structure to which they point.

The code format for PA-RISC code will be **BIG ENDIAN**. The PCI header and PCI Data Structures will be **LITTLE ENDIAN**.

Below is the format of the PCI Header with HP unique fields added.

Note: Little Endian Format

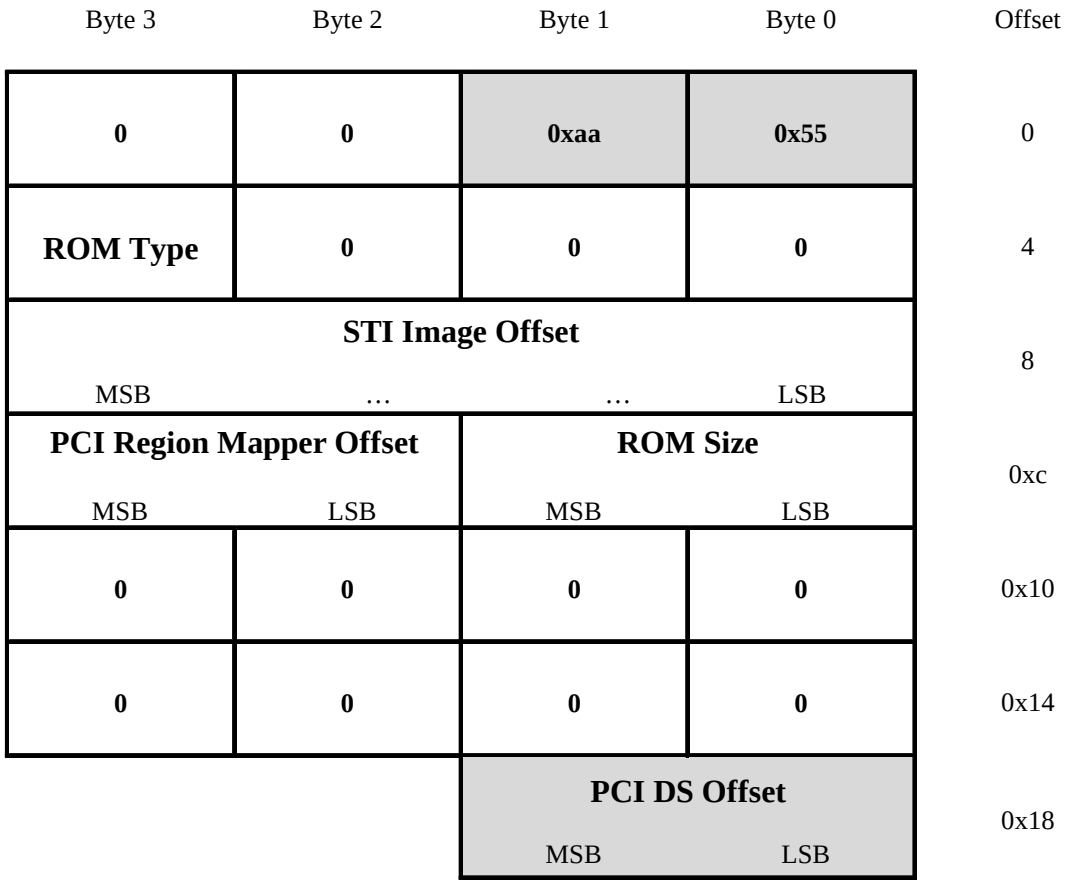

Figure 7 - PCI Header in Little Endian Format

Note: The shaded areas are architected by PCI 2.1, all other areas are HP Unique.

4.1.3.3.1 Header – PCI Architected

The “55Aah” is the option ROM signature, which is a x86 BIOS legacy, and indicates this to be a valid PCI ROM.

4.1.3.3.2 ROM Type – HP Unique

This number is valid only when the “code type” byte in the PCI Data structure indicates the code to be of type “PA-RISC”. See the details of the PCI DS.

Valid ROM Types are:

0 = Undefined ROM type image
1 = STI ROM type image

2 = PA-RISC IODC type image (PA1.1)
9-255 = Reserved for future PCI HP-PA hardware

4.1.3.3.3 STI Image Offset – HP Unique

This is a 32 bit offset (in little endian format) from the beginning of the PCI ROM header to the first byte of the STI image.

4.1.3.3.4 ROM Size – HP Unique

This field indicates the size of the ROM. It is in multiples of 512 bytes.

4.1.3.3.5 PCI DS Offset – PCI Architected

This is a 16 bit offset (in little endian format) from the beginning of the PCI ROM header to the PCI Data Structure.

4.1.3.3.6 PCI Region Mapper Offset – HP Unique

This is a 16 bit value that is an offset from the beginning of the ROM to the beginning of the PCI region mapper array. Below is a description of the pci region mapper array.

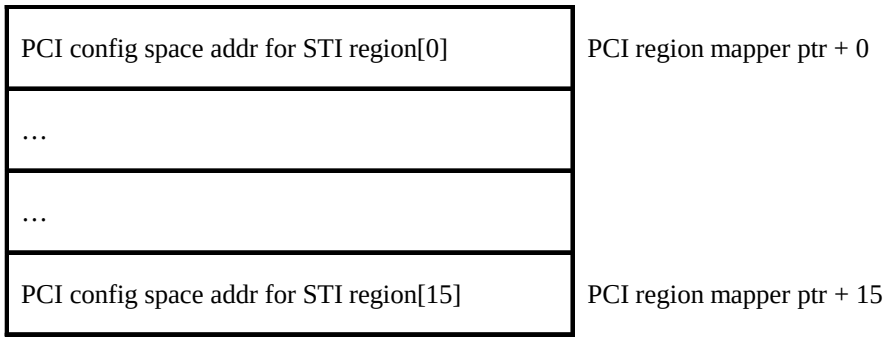

Figure 8 - PCI Region Mapper for STI

Each entry in the pci region mapper array is one byte. The value in each byte will be a pci config space offset. Normally this will be one of the pci bars, 0x10 for example. Unused entries contain 0.

To understand the need for this you will have to read the section on region mapper in the remaining document. STI breaks the physical card into 16 possible regions, each in a separate, non-contiguous space and the region mapper is the mechanism used to relate the pci bars to the STI regions.

Note that the STI dev region list is currently constrained to 8 entries but the pci region mapper can have up to 16 entries. This is for future expansion, currently the firmware and kernel only use 8 STI region entries, and so only the first 8 pci region mapper array entries will be used. The other 8 would contain 0.

The STI dev region list, as explained later, makes the assumption that the entries in the device region list are all offsets from the base of the card's hard physical address (HPA). This worked well for the SGC and GSC bus structures. But, for PCI, this was not sufficient. The PCI architecture lets you map various parts of the hardware basically anywhere you want, and it does this through the PCI BAR mechanism. Since STI has made the assumption that all regions are offset from the start of the hardware, the PCI region mapper was invented. This allows each individual STI device region to be an offset from a particular PCI BAR entry, and not from the base of the hardware.

4.1.3.3.7 PCI Data Structure – PCI Architected

There are no changes from the standard PCI Data Structure definition. The only addition is the PA-RISC code type assigned by PCISIG. The following code types are valid:

0	**=**	**Intel x86 PC-AT compatible code**
1	**=**	**Open Firmware for PCI**
0x10	**=**	**HP PA-RISC code**

Note: The PCI Spec Revision 2.2 (Dec 18, '98) indicates that 2 is assigned to HP PA-RISC code but this is incorrect. They assigned 0x10 (or maybe they assigned 2 but told us the wrong value), in any case, to work properly with System firmware and HP-UX kernel you must use 0x10 in this field.

See the PCI spec for further details.

Note: Little Endian Format

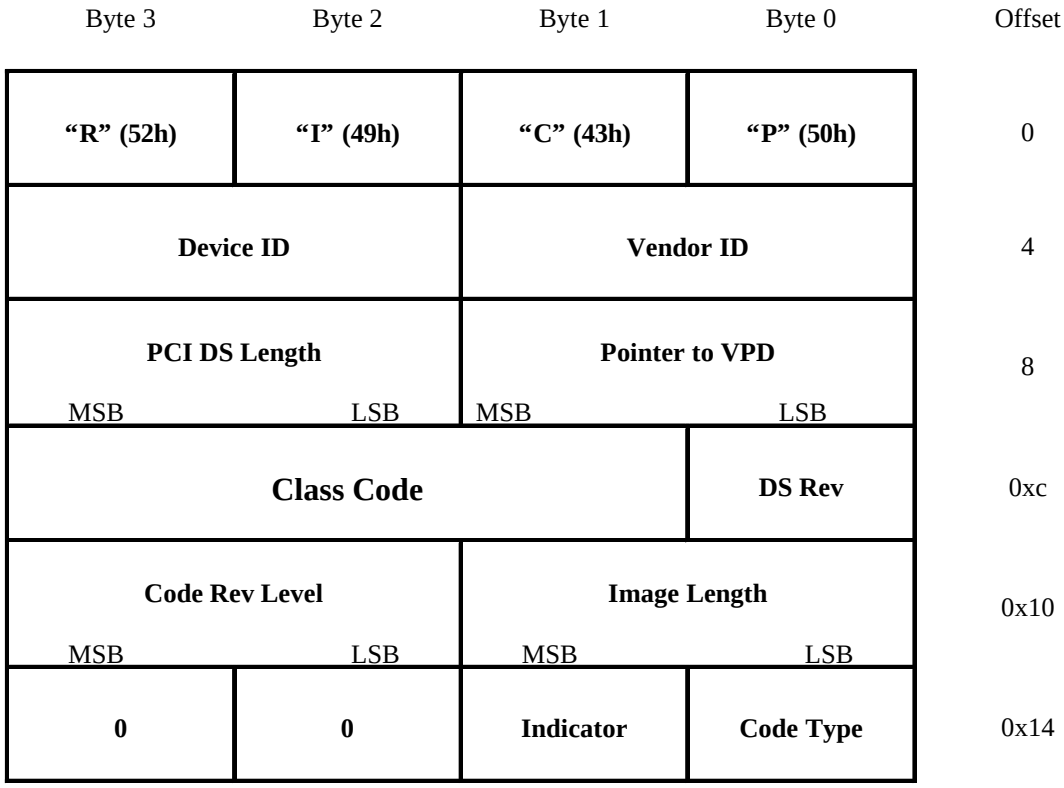

Figure 9 - PCI Data Structure

Note: All fields are architected by PCI spec rev 2.1

4.1.3.4 PCI dual EROM mapping

For SGC and GSC based graphics cards, the STI ROM must be available at the base HPA. For PCI based cards the STI ROM must be accessible through the PCI expansion ROM space and optionally, the base HPA. There is no longer a requirement to have the STI ROM available at the base HPA for PCI based devices, but this is SPU dependent and only the newer SPU will handle this.

4.1.4 Aberrations

4.1.4.1 Column mode

A word mode ROM accessed in a byte mode manner will yield a column mode ROM. Various data like fonts can be stored along the columns. The only reason this has been done in the past is when the hardware design was incorrect and did not properly decode the word mode ROM.

4.1.4.2 Romless (built into boot ROM)

Capability does exist to integrate the STI image with the BOOT ROM image, but this does require both FIRMWARE and OS (HP-UX) to agree on how access is made. This was done with the built in graphics on the 712 and on the later generation 715 and 725 products.

For these products, the firmware and kernel will find the pointer to the STI image at offset 0x604 in page zero. This is a 4 byte pointer to the location of the STI image within the BOOT ROM image.

4.2 STI Image Format

4.2.1 General

The STI ROM image is broken up into a Device Data section (which is architected) and the remainder of the ROM where all the routines, font tables live and various other useful parts.

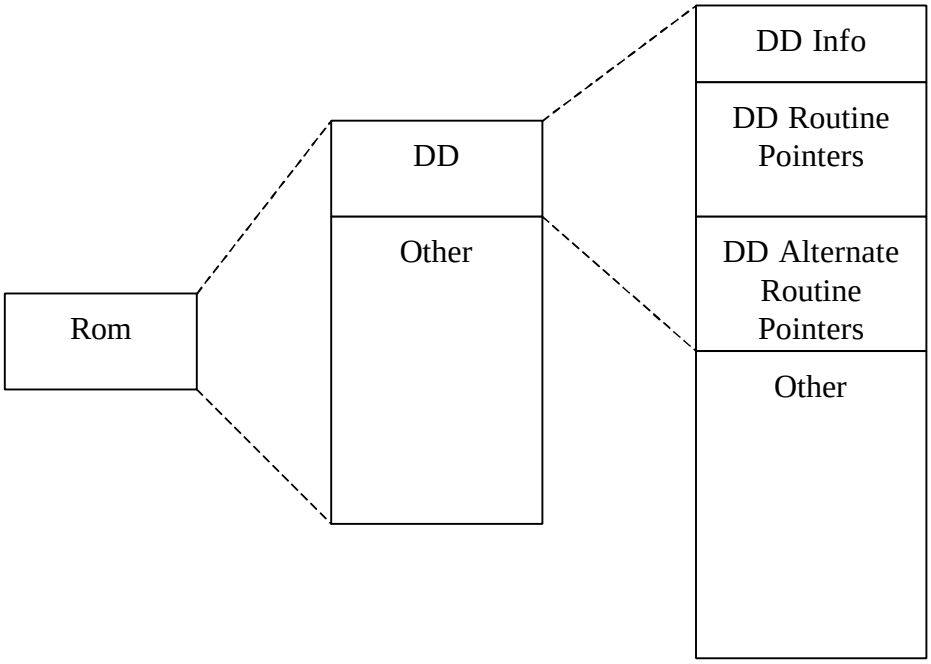

Figure 10 - STI ROM Structure

Note: For PCI, this is the STI Word mode image part of figure 4. Otherwise, this is either the physical ROM on the card or the image of the STI in the boot ROM.

The Device Data (DD) section contains offsets to the various architected fields in the ROM, such as the different routines and the beginning of the font tables. And is typified in the following figure.

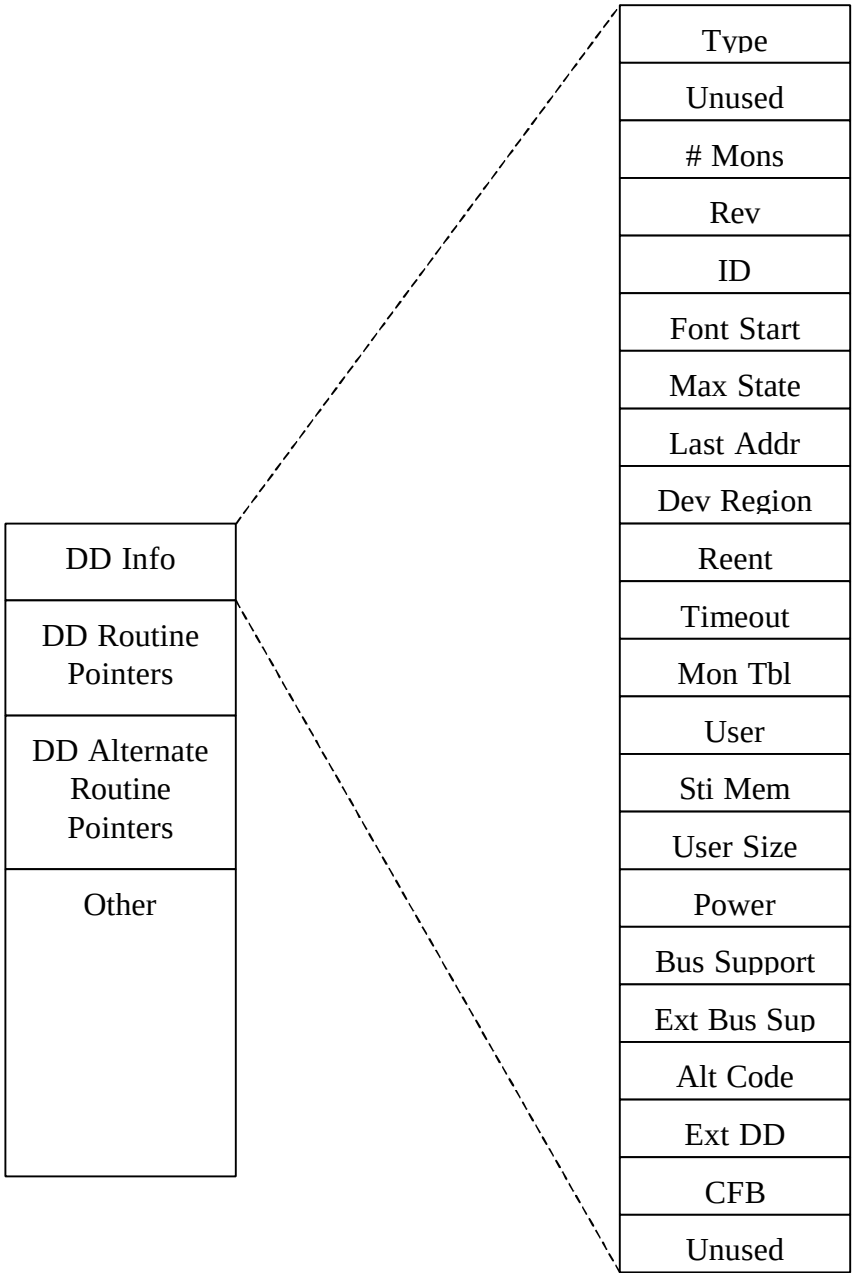

Figure 11 – DD Info Structure

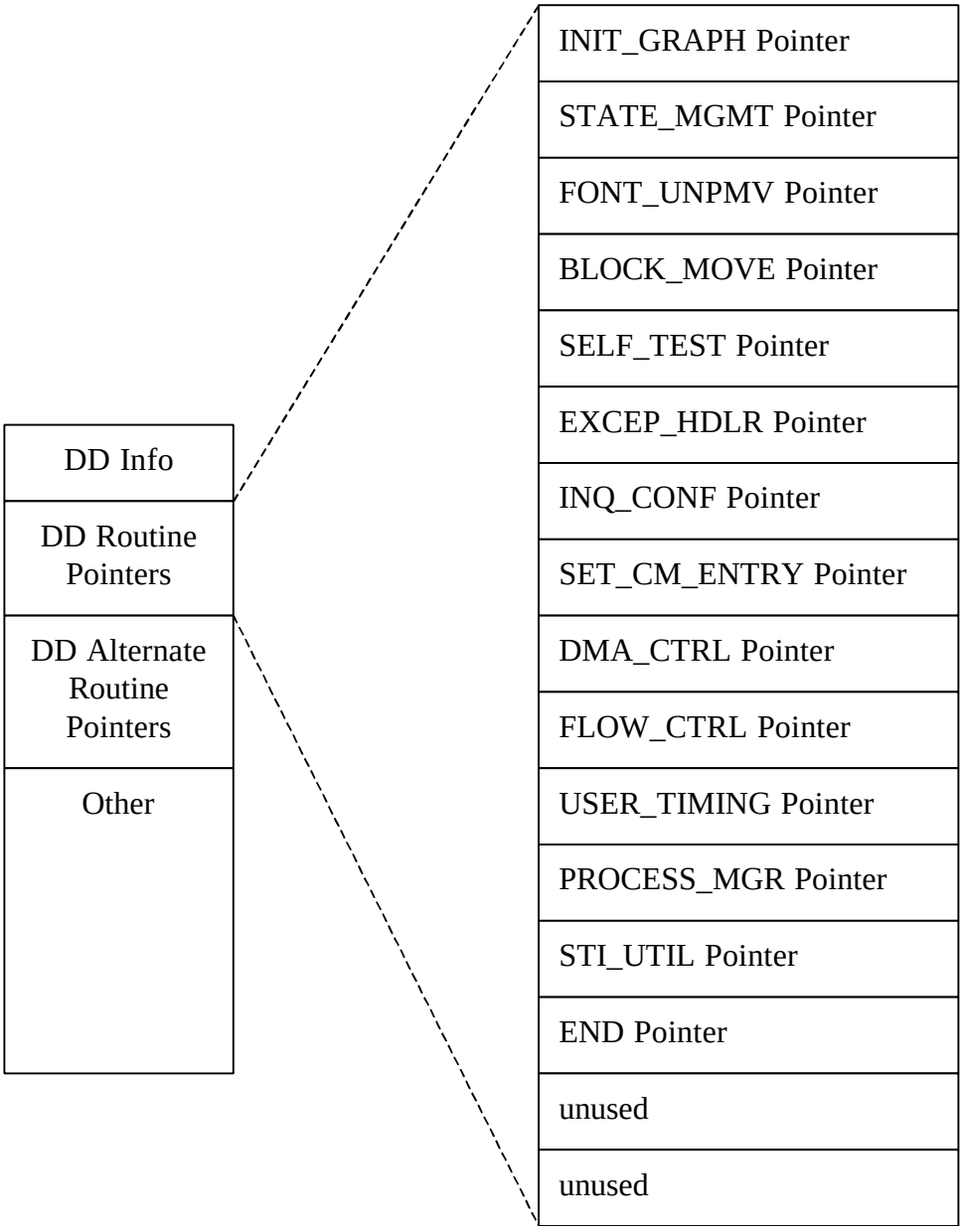

Figure 12 –Routine Structure

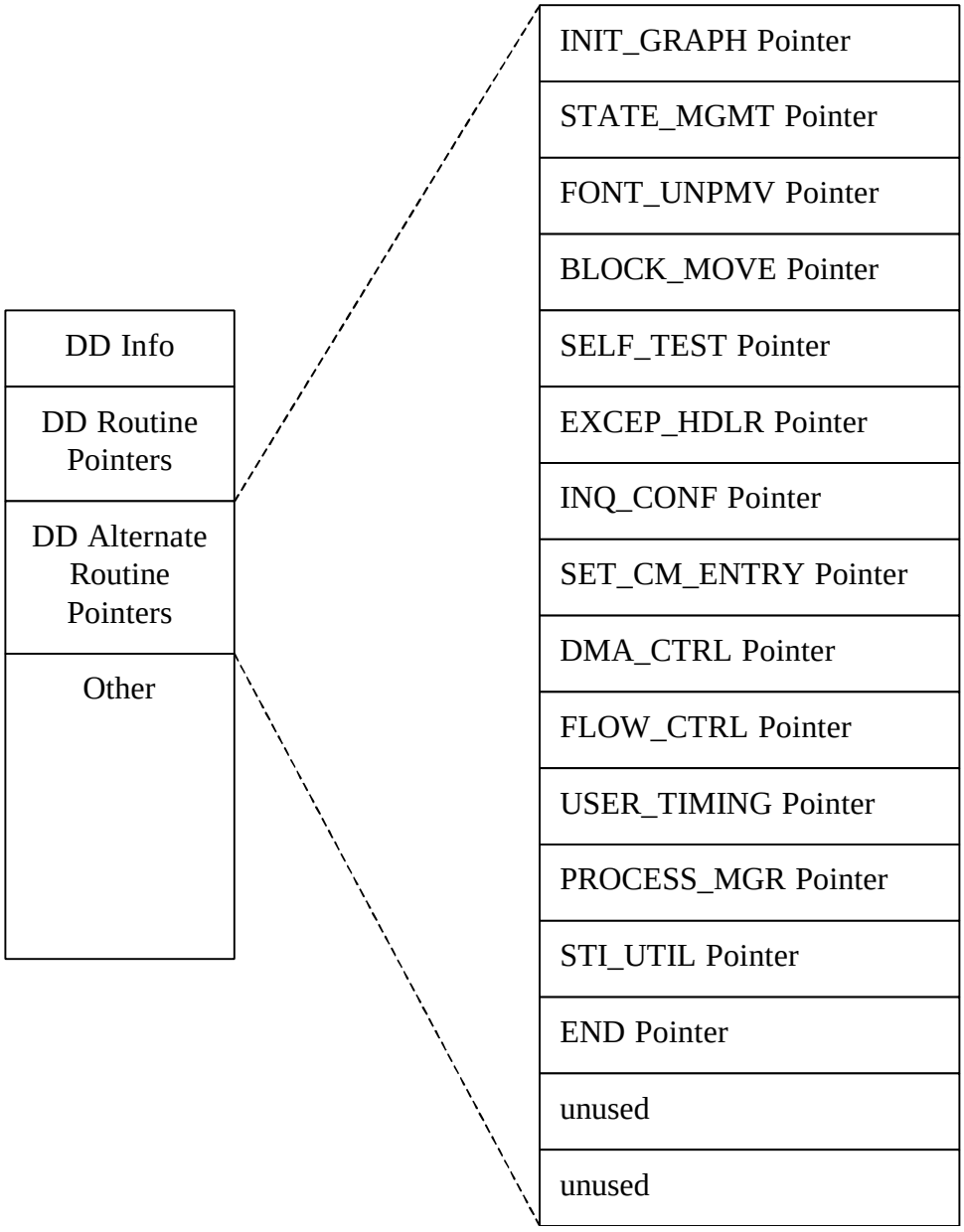

Figure 13 – Alternate Routine Structure

The beginning of the STI image contains the DD section. It contains information about the type of device present. A boot ROM searching for a graphics device would do so by reading byte 3 (LSB of the first word) at the low portion of each possible graphics device location STI ROM (base HPA for non PCI or PCI expansion ROM). A bus error would indicate no graphics device is present. If a graphics device is found in a given slot, self test must still be run on the device before it is used to insure that it is fully functional.

The initialization routine must be run both before and after any self test as self test can leave the device in an unknown state.

The addressing scheme in the STI ROM uses a format whereby addresses contained in the DD are all 32 bits and are byte counts relative to the starting address of the device. (For SGC devices all but the top (most-significant) 6 bits must contain zeroes.) The absolute location can be determined by the SPU by substituting the top 6 bits of the graphics board address (i.e., its SGC slot address) into the corresponding 6 bits of the address read from the DD.

> *NOTE: This is not true for PCI based systems. The older SGC and GSC based systems required the graphics to be at 0xf400000, 0xf6000000, 0xf8000000 or 0xfa000000. There is no such requirement on PCI based graphics.*

It is assumed that the SPU will have a separate I/O space for devices with STI ROM's and will know that only such devices can exist there.

The DD uses 768 bytes of mapped byte mode ROM space. A tabular description of the complete DD section follows the description of each section.

Field Name	Word Mode Address	Word Mode Size (Bytes)	Byte Mode Address	Byte Mode Size (Bytes)
Device Type	00	1	-	-
Device Type	01	1	-	-
Device Type	02	1	-	-
Device Type	03	1	03	1
Unused	04	1	-	-
Number Monitors	05	1	07	1
Global Rom Rev	06	1	0b	1
Local Rom Rev	07	1	0f	1
Graphics ID (MSB)	08	8	13	8
Graphics ID	09		17	

Graphics ID	0a		1b	
Graphics ID	0b		1f	
Graphics ID	0c		23	
Graphics ID	0d		27	
Graphics ID	0e		2b	
Graphics ID (LSB)	0f		2f	
Font Start Addr (MSB)	10	4	33	4
Font Start Addr	11		37	
Font Start Addr	12		3b	
Font Start Addr (LSB)	13		3f	
Max State Storage (MSB)	14	4	43	4
Max State Storage	15		47	
Max State Storage	16		4b	
Max State Storage (LSB)	17		4f	
Rom Last Addr (MSB)	18	4	53	4
Rom Last Addr	19		57	
Rom Last Addr	1a		5b	

Rom Last Addr (LSB)	1b		5f	
Device Region List (MSB)	1c	4	63	4
Device Region List	1d		67	
Device Region List	1e		6b	
Device Region List (LSB)	1f		6f	
Max Reent Storage (MSB)	20	2	73	2
Max Reent Storage (LSB)	21		77	
Max Timeout (MSB)	22	2	7b	2
Max Timeout (LSB)	23		7f	
Mon Table Address (MSB)	24	4	83	4
Mon Table Address	25		87	
Mon Table Address	26		8b	
Mon Table Address (LSB)	27		8f	
User Data Address (MSB)	28	4	93	4

User Data Address	29		97	
User Data Address	2a		9b	
User Data Address (LSB)	2b		9f	
STI Memory Request (MSB)	2c	4	A3	
STI Memory Request	2d		A7	
STI Memory Request	2e		Ab	
STI Memory Request (LSB)	2f		Af	
User Data Size (MSB)	30	4	B3	
User Data Size	31		B7	
User Data Size	32		Bb	
User Data Size (LSB)	33		Bf	
Power Usage (MSB)	34	2	C3	2
Power Usage (LSB)	35		C5	
Bus Support	36	1	Cb	1
Extended Bus Support	37	1	Cf	1

Alternate Code Type	38	1	D3	1
Extended DD Struct 1	39	1	D7	1
Extended DD Struct 2	3a	1	Db	1
Extended DD Struct 3	3b	1	Df	1
CFB Address (MSB)	3c	4	E3	
CFB Address	3d		E7	
CFB Address	3e		Eb	
CFB Address (LSB)	3f		Ef	
Unused	-	-	F3	4
Unused	-	-	F7	
Unused	-	-	Fb	
Unused	-	-	Ff	

Figure 14 - DD Structure

4.2.2 DD Struct Field Descriptions

4.2.2.1 device type

The Device Type is included to cover the possibility that non-graphics devices may use the STI ROM spec in the future. Currently, the only defined device types are one for Byte mode ROM's and 3 for Word mode ROM's. Device type zero is, and must remain, unused.

Note that in byte mode ROM's, the device type is the first valid byte in the ROM which is the least significant byte of the first word. In word mode ROM's the least significant byte still contains the actual device type, but that type is also copied into the other 3 bytes of the first word.

Remember that in byte mode ROM's, the contents of the top 3 bytes are indeterminate. Sometimes they might show up as zero and other times not. Some bus controllers will actually return what ever was last in the bus latches, and other bus controllers might actually duplicate the least significant byte into the other three. So, be safe, in byte mode ROM's only use the least significant byte.

4.2.2.2 num mons

The num mons field identifies how many different monitor selections the STI image supports. This is typically used for supporting different resolution/refresh rate combinations. There is no requirement that STI support more than a single monitor type but customers expect the capability of using different monitors.

4.2.2.3 global revisions

Including a global ROM revision number in the STI ROM makes it possible to identify the STI ROM spec revision that the ROM was generated from.

For instance, STI ROM spec revision 8.02 must use a 0x82 for the global revision in the ROM. The current version of the spec is 8.0d (8.13) so the Global revision in a ROM based on this spec would be 8.13.

The lower nibble in the global revision in the ROM will be converted to a two digit decimal number and will become the two decimal digits to the right of the decimal point. The upper nibble will be converted to a one or two digit decimal number (as appropriate) and become the decimal digit(s) to the left of the decimal point in the spec revision. For example, Global revision number 0xA3 will correspond to spec revision 10.03.

4.2.2.4 Local revisions

Including a Local ROM revision number in the STI ROM makes it possible to identify the various version of the ROM for each specific device. This field must start with revision zero and must be incremented whenever the ROM is changed. The local revision number should be used for such things as indicating that the code in the ROM has changed to fix a problem. STI code developers should maintain records regarding the significance of this revision level. This local revision number must be reset to zero whenever the global revision has been incremented.

4.2.2.5 graphics id

The graphics ID has been allocated 64 bits to allow the ID to be assigned via a scheme similar to that in use at the Apollo Division. In addition, the most significant 32 bits of the ID will also be unique from device to device. Users who require a 32 bit ID should

use only the upper half of the ID while 64 bit ID users should use the entire ID. Note that the least significant 32 bits will not be unique between devices. Appendix A discusses the means by which ID's can be acquired.

4.2.2.6 font start

These 4 bytes contain a byte count to the first valid byte of the font storage area in STI ROM, relative to start of the STI ROM. All addresses in the font storage space are relative to the start of the font storage and need this value added to them to get absolute addresses. The STI ROM must provide at least one font although this font may vary in character size from device to device. An option is provided in the font storage area for adding additional fonts in ROM and providing a pointer to the start of the next font. As with all other areas of storage in the ROM, the font storage area only uses the least significant byte of each word.

4.2.2.7 Maximum state storage

This location contains the number of 32 bit words of RAM for state storage that must be made available to the state management routine if state will be saved and restored. The contents of this space must be preserved if a state needs to later be restored.

4.2.2.8 Last address of ROM

These four bytes contain the byte count to the last valid byte in the STI ROM, relative to the start of the STI ROM. This value is needed by the routine doing the CRC check of the ROM.

The values placed at the last two addresses are those which would set a CRC of the entire STI ROM space to zero.

The ROM's first valid word is at relative location 0x000000 of the graphics device’s space, and the last valid word is at the "End of ROM Address". Special care must be taken to observe the ROM's single-byte width so only valid bytes are included in the CRC. For word mode ROM all 4 bytes are valid and must be included as part of the CRC.

Performing a CRC on the ROM using the algorithm described in Appendix B can test the STI ROM. A CRC is being used instead of a checksum, as it is able to detect more error conditions than a simple checksum. If the result is zero, the CRC is correct.

4.2.2.8.1 crc depends on ROM an even multiple of bytes

4.2.2.9 dev region list

These four bytes contain a pointer to a table within the STI ROM. This table contains the necessary information to map the various regions of this device into virtual memory. The structure needed for this table is detailed in Appendix C. Each table entry contains the following data. Due to size limits in current kernel structure, this region list may contain a maximum of eight entries.

- A 14 bit value (offset) which indicates what the offset (in 4 Kbyte pages) is, from the start of the device's I/O space to the start of the region to be mapped.

- A one bit value indicating whether the user should be allowed access to this region (sys_only=1) or not (sys_only=0).

- A one bit value indicating whether this region should be mapped into data cache, if possible, (cache=1) or not (cache=0). If not possible, regions with this bit set should be mapped as uncached.

- A one bit value indicating whether this region should be mapped in using a block TLB if a block TLB is available (btlb=1) or not (btlb=0).

- A one bit value indicating whether this is the last region in the list (last=1) or not (last=0).

- A 14 bit value (length) which indicates how many 4 Kbyte pages must be mapped in for this block.

The region list structure in the STI ROM also contains a *future ptr which can be used to expand the size or contents of the region list if needed in the future. This pointer must be the next entry after the last valid region entry and must be set to zero if unused.

By definition, region zero must be the STI ROM, region one must be the graphics device’s framebuffer and region two must be the graphics device's control space. The STI ROM must exist. If either of two other regions does not exist, then the region list must contain an entry for that region but the offset and length fields must both be set to zero. Unused regions can exist in the table as valid regions but must be indicated by an entry of zero for the offset and length.

For example, if we have a very simple graphics device that contains only a STI ROM and framebuffer, the region list must contain the following data. The first entry contains the offset and length for the ROM region and the "last" bit cleared to indicate that it is not the last entry. The second entry contains the offset and length for the framebuffer and also has the "last" bit cleared to indicate that it is not the last entry. The third entry has a zero for both offset and length to indicate that there is nothing to map in for the control space. The third entry has the "last" bit set to indicate that this is the last entry. In this way, the table has indicated that we have three regions (the minimum required), but that only two of them have any space that needs to be mapped in.

All the other regions (to a maximum of eight total regions) are device dependent and their assignments must be defined in each device's External Reference Specification (ERS). Although this region list may have up to eight regions defined, the actual GCDESCRIBE, ioctl system call can only return a maximum of six device-dependent regions in a region list. The framebuffer and graphics control regions are returned as separate parameters. The translation between the region list used by the STI ROM and the region list returned by ioctl(GCDESCRIBE) is given below. The five device dependent regions must be organized such that the regions that can be mapped into user space are first, followed by the regions which can only be mapped into system space. For example if we have three total regions, two of which can be mapped into user space, then these two regions must be device dependent regions 1 and 2. The system region would then be device dependent region 3.

All the regions in the region list must have physically distinct addresses. That is, none of the regions may have the same or overlapping physical addresses.

Region Name	STI Reg no.	GCDESCRIBE crt_region[N]
STI ROM	0	0
Framebuffer	1	(crt_frame_base)
Control Space	2	(crt_control_base)
Device Dep reg 1	3	1
Device Dep reg 2	4	2
Device Dep reg 3	5	3
Device Dep reg 4	6	4
Device Dep reg 5	7	5

Figure 15 – HP-UX Region Usage

This table is used by the calling process (e.g., kernel or boot ROM) by reading its contents and mapping each region into its memory space (virtual or physical), as appropriate. An array containing the addresses of the mapped in regions is then created and put into the glob conf structure by the calling routine. If any of the eight possible regions has not been mapped in, then that entry in the array must be null. In its simplest usage, the region array in the glob conf structure would be created from the region table

in the STI ROM as follows. The calling routine would take the 14 bit offset value from the ROM table and put it in the lower 14 bits of a 32 bit word. This value would then be shifted left 12 bits to change it from a 4 kbyte page count to a byte count. The starting address of this device's I/O space would then be added to this value to get the starting address of this region. This would be done for each of the valid regions in the ROM table.

The block TLB (see below) flag should be used in the following manner. The STI ROM developer should flag all regions that could significantly benefit from being mapped with a block TLB, with the btlb flag equal to one. STI ROM developers should consult with graphics driver developer and kernel developers to understand the best usage of block TLB's. Calling routines that are using the block TLB flag should implement the following algorithm for mapping in regions with block TLB's. The calling routine should first determine if the framebuffer (region 1) is marked for mapping with a block TLB. If so, the calling routine should scan all the other regions to identify those regions which are flagged for block TLB mapping and which exist in the same block TLB boundaries (e.g., 16Mbytes) as the framebuffer. The calling routine should then map in all these regions using the first available block TLB. If other block TLB's are available for use by this device, the calling routine should scan the remaining regions for those regions which are flagged for block TLB mapping but weren't mapped with the first block TLB. These regions should be grouped as was done for the first block TLB and then mapped. Note that the first block TLB should be used for the region containing the framebuffer. It is expected that few, if any, graphics devices will be able to effectively use more than two block TLB's and rarely will a device have access to more than two block TLB's.

4.2.2.9.1 Block TLBs

New SPU/OS architectures are beginning to support a Translation Lookaside Buffer (TLB) which allows a block of a device's physical addresses to be mapped into virtual memory as a very large block rather than only in a 4kbyte blocks. There are significant performance advantages to mapping in a graphics device's I/O space in this manner but there are usually constraints that must be met to allow it to be done. For instance on PCX-S (the snakes chip set), in order for a 16 Mbyte block to be mapped into user space, it must be a contiguous block of user space and must be on a 16 Mbyte boundary. Hardware designers are strongly encouraged to discuss the planned memory map for their devices with the OS developers early enough to allow changes to be made if necessary. In addition, the STI code developers need to provide a regions list that will allow the kernel to conclude where it can create a large block out of several regions without having any device dependent information other than access to the region list in the STI ROM. This may require the size of certain regions in the regions list to be expand to include unused areas of the memory map.

4.2.2.10 max reent

These two bytes contain the number of words which must be provided by the calling routine if any STI routines are asked to exit (WAIT=0) rather than busy wait. The save addr pointer in glob_conf points to the start of this space. This space is global and therefore its contents must be preserved between repeated calls to an STI routine that has not completed. If the calling routine allows busy waiting (WAIT=1), then the save_addr pointer must be null and no space is therefore provided.

4.2.2.11 max timeout

These two bytes contain a count of the maximum number of tenths of seconds which any of the STI routines could take from when it is started to when it would normally finish. This count should be used by calling routines to determine if the current STI routine has hung and should be aborted. Any STI routine that takes more than this time to complete should be considered as hung. It is expected that this timer value would be used with a hardware timer in the SPU that could issue an interrupt after the time had elapsed.

4.2.2.12 mon tbl

For graphics devices that support multiple monitors this entry identifies the number of monitors supported.

4.2.2.13 user data offset

Data that is used both by the STI firmware as well as user mode code can be located by using user data. This has typically been done for timing data so that user mode code such as SAM can access the timing parameters and display a dialog to the user on timing choices.

4.2.2.14 sti mem request

Since each STI routine is independent of every other routine, it is sometimes necessary for routines to be able to pass data between themselves. This indicates the number of bytes required by STI. Note that current SPU firmware limits the maximum amount to 256 bytes, but the kernel has no such limitation.

4.2.2.15 user data size

This is the size of the data pointed to by the user data offset.

4.2.2.16 Power Usage

Number of Watts (maximum) used by the graphics hardware. SPU firmware, as well as the kernel, to adjust internal fan speeds uses this value.

4.2.2.17 Bus support

The bus support byte contains various flags that indicate special support required for different bus architectures. Included below is the current definition for each of the bits.

```
#define BUS_SUPPORT_GSCINTL         0x01
#define BUS_SUPPORT_GSC15X          0x02
#define BUS_SUPPORT_GSC2X           0x04
#define BUS_SUPPORT_PCI_IOEIM       0x08
#define BUS_SUPPORT_IMPLICIT_LOCK   0x10
#define BUS_SUPPORT_PCI_DUAL_DECODE     0x20
#define BUS_SUPPORT_PCI_EROM_MMAP       0x40
#define BUS_SUPPORT_PCI_STD_INT         0x80
```

The GSCINTL is a representation that the card supports pulling the INTL line on the GSC bus to indicate an interrupt.

The GSC15X indicates the card supports the GSC1.5X spec, and the GSC2X indicates the card supports the GSC2X spec.

The PCI_IOEIM indicates the card will use the PA-RISC directed interrupt mechanism that will write specific data (eim_data) to a targeted processor at address (eim_addr).

The PCI_STD_INT indicates that the card will use the PCI int line to indicate an interrupt.

The IMPLICIT_LOCK indicates this card supports the implicit locking mechanism.

The PCI_DUAL_DECODE bit, when 0, indicates that there are 2 address decoders, one for the PCI expansion ROM and a separate one for the PCI bars. When this bit is 1 it indicates there is a single, shared, decoder. This is significant for both firmware and kernel because in PCI, only a single device on a PCI bus may have the PCI expansion ROM enabled at a single time. This means the caller will have to manage all other PCI Expansion ROM (disabling them) if there is a shared decoder, when accessing the STI ROM.

The PCI_EROM_MMAP field, when 0, indicates that the ROM is multiply mapped both in the PCI expansion ROM space and from one of the PCI bars. When this field is 1 it indicates that the ROM is available only through the PCI expansion ROM space and not through one of the PCI bars.

4.2.2.18 ext bus support

The extended bus support field is an additional byte for more bus specific information. Included below is the current definition of this byte.

#define EXT_BUS_SUPPORT_DMA 0x01
#define EXT_BUS_SUPPORT_IMPLICIT_LOCK_PIO 0x02

The SUPPORT_DMA bit, when set, indicates the card supports DMA.

The SUPPORT_IMPLICIT_LOCK_PIO bit, when set, indicates the card supports implicit locking, but only through programmed I/O and not via DMA.

4.2.2.19 alt code type

The Alternate code type byte indicates the type of code that will be found in the Alternate DD Routine Pointers area.

#define ALT_CODE_TYPE_UNKNOWN 0x00
#define ALT_CODE_TYPE_PA_RISC_64 0x01

The only usable value is 1 which indicates that the routines pointed to in the Alternate Routines Pointer area are for 64bit PA-RISC processors.

4.2.2.20 ext dd struct

This is a place holder for future extensions to the DD struct. It is currently unused and should be zero. In the future, this would be an offset to more DD struct data.

4.2.2.21 cfb offset

STI includes the capability of containing the "Color Frame Buffer" driver for the X11 server. The purpose of this driver is to allow a firmware driver to be used for X11 until the system can be updated with the accelerated X11 driver. This is useful for new installs of the OS.

4.2.3 mon tbl descriptor strings

Monitor Table descriptor strings is used only by SPU hardware and provides a text string that defines a particular monitor setting. System firmware, when commanded, can use this string to present a choice to the user on monitor selection.
This pointer will point to an array of strings. The number of strings in this array is equal to the num_mons field. The array is not necessarily null terminated, only each string. Calling code must know the number of strings in the array and stop looking after that number has been reached.

4.2.4 User Data

Monitor Table descriptor strings is used only by SPU hardware and provides a text string that defines a particular monitor setting. System firmware, when commanded, can use this string to present a choice to the user on monitor selection.
This pointer will point to an array of strings. The number of strings in this array is equal to the num_mons field. The array is not necessarily null terminated, only each string. Calling code must know the number of strings in the array and stop looking after that number has been reached.

4.2.5 Overlay Routines

Monitor Table descriptor strings is used only by SPU hardware and provides a text string that defines a particular monitor setting. System firmware, when commanded, can use this string to present a choice to the user on monitor selection.
This pointer will point to an array of strings. The number of strings in this array is equal to the num_mons field. The array is not necessarily null terminated, only each string. Calling code must know the number of strings in the array and stop looking after that number has been reached.

4.2.6 Regions

Monitor Table descriptor strings is used only by SPU hardware and provides a text string that defines a particular monitor setting. System firmware, when commanded, can use this string to present a choice to the user on monitor selection.
This pointer will point to an array of strings. The number of strings in this array is equal to the num_mons field. The array is not necessarily null terminated, only each string. Calling code must know the number of strings in the array and stop looking after that number has been reached.

4.2.7 Device Dependent Routines

In order to meet the objective of decoupling kernel and boot ROM releases from graphics hardware releases, it is necessary to define a standard interface between the SPU and graphics hardware. Providing a set of routines in the STI ROM which contain device dependent information, but which the SPU can use in a device independent manner does this. These routines are described below. Each of these routines will provide the capability to have arguments passed into and returned from them.

It is also necessary to provide each graphics device with the ability to work without change on either a 68000 based SPU or a PA-RISC based SPU. To accomplish this, the STI ROM will contain a set of routines in 68000 code and a set of routines in PA-RISC code. It is believed that this approach is more cost effective than using pseudo-code and requiring the SPU to incorporate an interpreter in a ROM. To support these two sets of routines, the DD portion of the ROM is partitioned in the following manner. Pointers (byte counts to the first valid byte) to PA-RISC routines start at location 0x0103 and can extend through location 0x01ff. Pointers (byte counts to the first valid byte) to 68000

routines start at 0x0203 and can extend through location 0x02ff. The pointers to the routines are byte addresses of the valid byte in ROM (i.e., the same format as the DD section of the ROM). The upper six bits of these pointers must contain zeros and the addresses are all relative to the start of this device graphics space. Remember that the DD portion of the ROM is always organized in a byte wide structure.

Note: The Motorola based SPU product is no longer supported and the Alternate Routine DD Struct pointers have been re-architected to be indicated as to code type by using the ALT_CODE_TYPE field in the DD struct, as previously explained.

Field Name	Word Mode Offset	Byte Mode Offset	Bytes
INIT_GRAPH (MSB)	0x40	0x103	4
INIT_GRAPH		0x107	
INIT_GRAPH		0x10b	
INIT_GRAPH (LSB)		0x10f	
STATE_MGMT (MSB)	0x44	0x113	4
STATE_MGMT		0x117	
STATE_MGMT		0x11b	
STATE_MGMT (LSB)		0x11f	
FONT_UNPMV (MSB)	0x48	0x123	4
FONT_UNPMV		0x127	
FONT_UNPMV		0x12b	
FONT_UNPMV (LSB)		0x12f	
BLOCK_MOVE (MSB)	0x4c	0x133	4
BLOCK_MOVE		0x103	
BLOCK_MOVE		0x13b	
BLOCK_MOVE (LSB)		0x13f	

SELF_TEST (MSB)	0x50	0x143	4
SELF_TEST		0x147	
SELF_TEST		0x14b	
SELF_TEST (LSB)		0x14f	
EXCEP_HDLR (MSB)	0x54	0x153	4
EXCEP_HDLR		0x157	
EXCEP_HDLR		0x15b	
EXCEP_HDLR (LSB)		0x15f	
INQ_CONF (MSB)	0x58	0x163	4
INQ_CONF		0x167	
INQ_CONF		0x16b	
INQ_CONF (LSB)		0x16f	
SET_CM_ENTRY (MSB)	0x5c	0x173	4
SET_CM_ENTRY		0x177	
SET_CM_ENTRY		0x17b	
SET_CM_ENTRY (LSB)		0x17f	
DMA_CTRL (MSB)	0x60	0x183	4
DMA_CTRL		0x187	
DMA_CTRL		0x18b	
DMA_CTRL (LSB)		0x18f	
FLOW_CTRL (MSB)	0x64	0x193	4
FLOW_CTRL		0x197	
FLOW_CTRL		0x19b	
FLOW_CTRL (LSB)		0x19f	

USER_TIMING (MSB)	0x68	0x1a3	4
USER_TIMING		0x1a7	
USER_TIMING		0x1ab	
USER_TIMING (LSB)		0x1af	
PROCESS_MGR (MSB)	0x6c	0x1b3	4
PROCESS_MGR		0x1b7	
PROCESS_MGR		0x1bb	
PROCESS_MGR (LSB)		0x1bf	
STI_UTIL (MSB)	0x70	0x1c3	4
STI_UTIL		0x1c7	
STI_UTIL		0x1cb	
STI_UTIL (LSB)		0x1cf	
END_ADDR (MSB)	0x74	0x1d3	4
END_ADDR		0x1d7	
END_ADDR		0x1db	
END_ADDR (LSB)		0x1df	
Unused	0x78	0x1e3	4
Unused		0x1e7	
Unused		0x1eb	
Unused		0x1ef	
Unused	0x7c	0x1f3	4
Unused		0x1f7	
Unused		0x1fb	
Unused		0x1ff	

Figure 16 - DD Routine Pointers for PA-RISC Code

Field Name	Word Mode Offset	Byte Mode Offset	Bytes
INIT_GRAPH (MSB)	0x80	0x203	4
INIT_GRAPH		0x207	
INIT_GRAPH		0x20b	
INIT_GRAPH (LSB)		0x20f	
STATE_MGMT (MSB)	0x84	0x213	4
STATE_MGMT		0x217	
STATE_MGMT		0x21b	
STATE_MGMT (LSB)		0x21f	
FONT_UNPMV (MSB)	0x88	0x223	4
FONT_UNPMV		0x227	
FONT_UNPMV		0x22b	
FONT_UNPMV (LSB)		0x22f	
BLOCK_MOVE (MSB)	0x8c	0x233	4
BLOCK_MOVE		0x203	
BLOCK_MOVE		0x23b	
BLOCK_MOVE (LSB)		0x23f	
SELF_TEST (MSB)	0x90	0x243	4
SELF_TEST		0x247	
SELF_TEST		0x24b	

SELF_TEST (LSB)		0x24f	
EXCEP_HDLR (MSB)	0x94	0x253	4
EXCEP_HDLR		0x257	
EXCEP_HDLR		0x25b	
EXCEP_HDLR (LSB)		0x25f	
INQ_CONF (MSB)	0x98	0x263	4
INQ_CONF		0x267	
INQ_CONF		0x26b	
INQ_CONF (LSB)		0x26f	
SET_CM_ENTRY (MSB)	0x9c	0x273	4
SET_CM_ENTRY		0x277	
SET_CM_ENTRY		0x27b	
SET_CM_ENTRY (LSB)		0x27f	
DMA_CTRL (MSB)	0xa0	0x283	4
DMA_CTRL		0x287	
DMA_CTRL		0x28b	
DMA_CTRL (LSB)		0x28f	
FLOW_CTRL (MSB)	0xa4	0x293	4
FLOW_CTRL		0x297	
FLOW_CTRL		0x29b	
FLOW_CTRL (LSB)		0x29f	
USER_TIMING (MSB)	0xa8	0x2a3	4
USER_TIMING		0x2a7	
USER_TIMING		0x2ab	

USER_TIMING (LSB)		0x2af	
PROCESS_MGR (MSB)	0xac	0x2b3	4
PROCESS_MGR		0x2b7	
PROCESS_MGR		0x2bb	
PROCESS_MGR (LSB)		0x2bf	
STI_UTIL (MSB)	0xb0	0x2c3	4
STI_UTIL		0x2c7	
STI_UTIL		0x2cb	
STI_UTIL (LSB)		0x2cf	
END_ADDR (MSB)	0xb4	0x2d3	4
END_ADDR		0x2d7	
END_ADDR		0x2db	
END_ADDR (LSB)		0x2df	
Unused	0xb8	0x2e3	4
Unused		0x2e7	
Unused		0x2eb	
Unused		0x2ef	
Unused	0xbc	0x2f3	4
Unused		0x2f7	
Unused		0x2fb	
Unused		0x2ff	

Figure 17 - DD Routine Pointers for Alternate Code

4.2.8 Font Storage

The STI ROM must also include at least a single font for use during boot up and for initial ITE support. The device dependent routines in the STI ROM will only support fonts of the same format as supplied in the ROM. The STI ROM must only provide a single font and is only required to provide the set of 128 standard ASCII characters.

Each font may have more than 128 characters in them to allow for the capability of using other fonts. If other fonts or more characters are provided then fonts must be provided for all the characters needed to insure a consecutive set of characters is present in each font.

It is also strongly recommended that the 128 character Katakana font be provided in the STI ROM

4.2.8.1 Multiple

4.2.8.2 Format

Because the STI ROM only allows a byte wide format, the font storage area must be organized to only use every fourth byte. If additional font storage area are provided by the applications and used by the STI ROM routines, they must also only use every fourth byte.

Inverse video is supported in the STI routines by the user calling the routines with the foreground and background color values interchanged.

Underlining is supported in the following manner. The standard font table supplied by the STI ROM will also contain the following information.

A value indicating how many pixels high the underline should be. For small fonts this will typically be one.

- A value indicating how far down (in pixels) from the top left corner of the font, the top left corner of the underline should be.

Underlining would be accomplished as follows. The character would first be written normally. The BLOCK_MOVE routine would then be called to perform a clear to the background color in the region where the underline should appear. Note that the clear functionality in BLOCK_MOVE performs a clear to the background color. Therefore, in order to get an underline in the foreground color, the background color must be set to the foreground color before the clear is performed. This region is defined by the width of the font and the height of the underline, and located using the offset provided. The height of the underline must be applied from the offset downward.

The font provided in the STI ROM is stored in the following format. The first row of pixels for a character is stored in as many bytes as needed, followed by the second row, etc.

If only a partial byte is needed to complete a row of pixels, then the pixels must be left justified within the byte. For example, a font that is10 pixels wide would use two bytes for the first row. The first byte would have valid pixels in all 8 bits while the second byte would only have valid pixels in the two left most bits.

In addition, the intercharacter space must also be included in the font. These spaces must be incorporated into the font such that they appear along the right and bottom edge of each character. At least a single pixel is needed. As an example, a 10x20 font would only contain a 9x19 block of data where pixels could be turn on. Blank spaces must also be provided above and below those characters which are not full height or which lack descenders.

If additional fonts are provided in the ROM, the four byte value provides an offset to the start of the next font. The value is an offset from the font storage area pointer contained in the DD portion of the ROM. If this offset is zero, then there are no more fonts in the ROM storage area.

The routine that unpacks and moves characters out of the font storage area can support other user supplied fonts if they are stored in an identical format. The font unpack routine must be able to support fonts of arbitrary size.

Field Usage	Word Mode Offset	Byte Mode Offset	Bytes
ASCII Code for first char in font (MSB)	0x0	0x03	2
ASCII Code for first char in font (LSB)		0x07	
ASCII Code for last char in font (MSB)	0x2	0x0b	2
ASCII Code for last char in font (LSB)		0x0f	
Width of Font in pixels	0x4	0x13	1
Height of Font in pixels	0x5	0x17	1
Font Type	0x6	0x1b	1

Bytes/Char (int((W+7)/8))*H	0x7	0x1f	1
Offset to start of next font (MSB)	0x8	0x23	4
Offset to start of next font		0x2f	
Offset to start of next font		0x2b	
Offset to start of next font (LSB)		0x2f	
Height of underline in pixels	0xc	0x33	1
Offset of underline	0xd	0x37	1
Unused	0xe	0x3b	1
Unused	0xf	0x3f	1
First Character	0x10	0x43	(int((W+7)/8))*H
Second Character			(int((W+7)/8))*H

Last Character			(int((W+7)/8))*H

Figure 18 - Font Structure

4.2.8.3 Types

The following fonts can be identified by setting the Font type field in font storage to the appropriate number.

Font	Font Type

Undefined	0
HPROMAN8	1
KANA8	2

Figure 19 - Font Types

4.3 Detailed Examination

4.3.1 PCI

What follows is an examination of a particular ROM dump. The following was dumped out of a raw ROM image. The byte offset is indicated by the number in the left most column, and each row contains 16 bytes. This is a hex dump so the second row starts at byte offset 16 decimal, the third row at bytes offset 32 decimal, etc.

One way to get this image would be do perform PCI config cycles of the expansion ROM space (PCI Config 0x30).

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
0000010 0000 0000 0000 0000 1c00 0000 5043 4952
0000020 3c10 8b10 0000 1800 0000 0203 df00 0000
0000030 1080 0000 1810 1818 0000 0000 0000 0000
0000040 0000 0000 0303 0303 0006 8c13 35ac da16
0000050 09a0 2587 0001 7d10 0000 0064 0001 b531
0000060 0001 7cfc 0000 00fa 0001 bfbc 0000 0324
0000070 0000 0100 0000 014e 000a 8001 0100 0000
0000080 0000 0000 0000 085c 0000 2b18 0000 36d4
```

As you can see, I only dumped out the first 128 bytes of the ROM image. As you will learn when the PCI header is decoded, this ROM image is actually 128 Kbytes in size. We don't need the entire ROM contents dumped to get a handle on the PCI header.

The first two bytes, 55aa, indicate this is a PCI Mode ROM.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

Since we have just identified this as a PCI Mode ROM, we know the next 5 bytes, starting at offset 0x2, are architected to be zero.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

The byte at offset 0x7, which is 0x1, indicates this is an STI image.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

The next four bytes starting at offset 0x8 are the STI Image pointer offset, and remember, this needs to be byte swapped, so the value is really 0x00000044.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

Since this is an offset from the start of the ROM image then we know that the STI image embedded within this PCI image starts at offset 0x44, from the base of the ROM image. Since this is a PCI image, we know that the STI must be a WORD mode image, and we can verify this by looking at the first four bytes starting at offset 0x44. The first four bytes within a word mode image indicate the type of STI image and for word mode must be equal to 0x03.

```
0000040 0000 0000 0303 0303 0006 8c13 35ac da16
```

We will take a closer look at decoding the STI image later.

The next two bytes starting at offset 0xC are the size of the STI image. Again, remember to byte swap this and multiply by 512. So, in this case, the value is 0x0100, or 256*512 which is 128 Kbytes.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

The next 2 bytes, starting at offset 0xe, is the offset to the PCI region mapper table. The value is 0x3400, which needs to be byte swapped, so the region mapper table starts at offset 0x0034, from the base of the ROM image.

```
0000000 55aa 0000 0000 0001 4400 0000 0001 3400
```

Since this is a PCI mode ROM image, we know the next two words, starting at offset 0x10 are architected to be zero,

```
0000010 0000 0000 0000 0000 1c00 0000 5043 4952
```

The last two bytes, starting at offset 0x18, are the offset to the PCI Data Struct, which in this case, after byte swapping, is 0x001c, from the base of the ROM image.

```
0000010 0000 0000 0000 0000 1c00 0000 5043 4952
```

Since the PCI Data Struct offset starts at 0x1c, the next two bytes, at offset 0x1a are unused and are zeroed out,

```
0000010 0000 0000 0000 0000 1c00 0000 5043 4952
```

Starting at offset 0x1c, which we determined to be the start of the PCI Data Struct, we can see the first four bytes are the "PCIR" identifier.

```
0000010 0000 0000 0000 0000 1c00 0000 5043 4952
```

The PCI Data Struct is 24 bytes in length and the only interesting part to decode is at offset 0x14, which is the code type field. Since the PCI Data Struct starts at 0x1c and we want the byte at offset 0x14 from there, we should be looking at 0x14+0x1c = 0x30. You will notice the byte at 0x30 is 0x10, which is the PCI assignment for PA-RISC code.

```
0000030 1080 0000 1810 1818 0000 0000 0000 0000
```

The decoding of the remainder of the PCI Data Struct is uninteresting so next we can examine the PCI region mapper table. Previously, it was determined that the PCI region mapper table began at offset 0x0034, and we know this will have 16 entries, each 1 byte in length so looking at these,

```
0000030 1080 0000 1810 1818 0000 0000 0000 0000
0000040 0000 0000 0303 0303 0006 8c13 35ac da16
```

Notice that the first 4 entries are non-zero, and the remainder are zero. Thus, there are only 4 of the region entries used. They indicate that STI region 0 is based from PCI Bar 0x18, STI region 1 is based from PCI Bar 0x10, STI region 2 is based from PCI Bar 0x18 and STI region 3 is based from PCI Bar 0x18.

This ROM image is for a card that uses two 64 bit regions. And for this card Bar 0x10 and Bar 0x14 make up one 64 bit address space and Bar 0x18 and 0x1c make up the other 64 bit address space. That is why you only see two different Bars used in this case.

4.3.2 Word Mode

This is a dump of a word mode STI image. This could have come from a PCI based card, where it would have followed the PCI header or it could be from a GSC based card, where word mode STI ROM is supported. It doesn't matter. The STI is independent of the PCI header part of the ROM.

Note that in the STI image part of the ROM everything is big endian, so there is no byte swapping needed in decoding the various pieces.

```
0000000 0303 0303 000f 8708 2d08 c0a7 09a0 2587
0000010 0000 6230 0000 0032 0000 a761 0000 6218
0000020 0000 00fa 0000 02c0 0000 03e0 0000 0100
0000030 0000 012c 0006 0200 0000 0000 0000 0000
0000040 0000 0890 0000 2fbc 0000 3a98 0000 4b54
```

```
0000050 0000 5210 0000 53f4 0000 5618 0000 5fdc
0000060 0000 6218 0000 0000 0000 0000 0000 0000
0000070 0000 0000 0000 0000 0000 0000 0000 0000
0000080 0000 6218 0000 6218 0000 6218 0000 6218
```

Word mode ROMs always start off with 4 identical bytes, each set to value 0x03,

```
0000000 0303 0303 000f 8708 2d08 c0a7 09a0 2587
```

The byte at offset 0x4 is unused and the byte following that, at offset 0x5, is the num_mons field, which in this case indicates 15 different monitors,

```
0000000 0303 0303 000f 8708 2d08 c0a7 09a0 2587
```

The byte at offset 0x4 is unused and the byte following that, at offset 0x5, is the num_mons field, which in this case indicates 15 different monitors

This will get you started in decoding the STI word mode image. Just follow along with the definition of the DD structure and you will not have any problems.

4.3.3 Byte Mode

This is a dump of a byte mode STI image.

Note that in the STI image part of the ROM everything is big endian, so there is no byte swapping needed in decoding the various pieces.

```
0000000 0000 0001 0000 0003 0000 0084 0000 0008
0000010 0000 002b 0000 00cb 0000 0001 0000 005a
0000020 0000 0009 0000 00a0 0000 0025 0000 0087
0000030 0000 0000 0000 0001 0000 00cb 0000 00f3
0000040 0000 0000 0000 0000 0000 0000 0000 0032
0000050 0000 0000 0000 0002 0000 00ac 0000 0077
0000060 0000 0000 0000 0001 0000 00cb 0000 0083
0000070 0000 0000 0000 0000 0000 0000 0000 00fa
0000080 0000 0000 0000 0000 0000 000b 0000 0003
0000090 0000 0000 0000 0000 0000 0000 0000 0000
```

As previously mentioned, if the first 4 bytes are 0x00000001 then the ROM is in the byte mode format. You will notice this is the case,

```
0000000 0000 0001 0000 0003 0000 0084 0000 0008
```

Since you know this is a byte mode ROM then you will always expect to see 3 leading zero bytes. Let's see how to decode the revision STI this ROM image contains. Looking

at the DD struct definition for the byte mode, we see that the global revision is at offset 0xb and the local revision is at offset 0xf,

```
0000000 0000 0001 0000 0003 0000 0084 0000 0008
```

At offset 0xb we see 0x84 and at offset 0xf we see 0x08. Thus the revision is 8.04/8, using the naming nomenclature of global/local.

This should be enough to get you started down the byte mode ROM decodes path.

5 Routines

5.1 general

5.1.1 calling conventions

The description of each routine also includes a section on the calling convention to which any calling routines interfacing to these routines must conform.

At one time we supported 68000 code as well as PA-RISC code, but since we no longer manufacture 68000 based workstation products we no longer support this capability. We now support 32 bit and 64 bit PA-RISC code. As mentioned previously, the 32 bit code is accessed via the DD Routines Pointers and the 64 bit code is accessed via the DD Alternate Routines Pointers. Of course, you must first check the ALTERNATE_CODE_TYPE entry in the DD Info structure, see Figure 14 - DD Structure, to verify that the code pointed to by the alternate pointers is 64 bit PA-RISC code.

It is expected that all the routines will be written in C and then compiled twice to get relocatable machine code. (Once for PA-RISC 32 bit and once for PA-RISC 64 bit). In this way, the interfaces to these routines will be standard C interfaces. See the Appendix for more information on the tools that are available to aid in this effort.

The calling conventions for each routine are similar in that they each pass four pointers to structures as parameters (all are passed even though all four structures may not be used). The exact structures for each routine are given in the Appendix.

- Flagptr - A pointer to the structure containing all the flags which may be passed IN or OUT of the routine. The contents of this structure do not need to be maintained between calls to this and other ROM routines. This structure also contains a pointer to allow future expandability while still maintaining compatibility.

- Inptr - A pointer to the structure containing the parameters which are passed into the routine. The contents of this structure do not need to be maintained between calls to this and other ROM routines. This structure also contains a pointer to allow future expandability while still maintaining compatibility.

- Outptr - A pointer to the structure containing the parameters which are passed out of the routine. The contents of this structure do not need to be maintained between calls to this and other ROM routines. This structure also contains a pointer to allow future expandability while still maintaining compatibility.

- Configptr - A pointer to the structure containing essential configuration information about the graphics device. This structure is global and must be saved between calls to

the various routines. The contents of this structure are limited to only those few configuration items that must be shared between routines. It is expected that generally the initialization routine will set the values in this structure and the other routines will use the contents. All routines must be passed the pointer to this space. This structure also contains a pointer to allow future expandability while still maintaining compatibility.

If the setting of multiple flags will result in a disallowed condition then the code must default to no action in all such cases.

In order to provide flexibility for possible future expansion, each structure includes an unused pointer (future_ptr). If additional parameters need to be added to any routine in the future, this pointer must be used to point to an additional structure containing the new parameters. When unused, this pointer must be null. It is necessary to include this pointer now to insure future compatibility for all the possible combinations in the future. For instance, suppose a new graphics device is developed which has added features that require more data to be passed to the INIT_GRAPH routine. This device's INIT_GRAPH routine would then use the future pointer to access this new data. However, to insure that this device will have predictable behavior with an old kernel that doesn't supply the new data, the INIT_GRAPH routine would need to see a null value for future pointer when used with an old kernel. In this way it can determine that the new data required by the device is not available and instead take some default action. Only in this manner can we truly decouple hardware and operating system releases while still providing the capability to add enhancements to the STI spec in the future.

If a new structure is added via the future pointer, the last item in the new structure must be another null future_ptr. In this way, further enhancements can continue to be added without impacting backward compatibility.

The general calling convention of each routine is as follows. The .h include file in the Appendix contains the typedefs for the structures.

```
int init_graph(flagptr,inptr,outptr,configptr)
        init_flags      *flagptr;
        init_inptr      *inptr;
        init_outptr     *outptr;
        glob_cfg        *configptr;

int state_mgmt(flagptr,inptr,outptr,configptr)
        state_flags     *flagptr;
        state_inptr     *inptr;
        state_outptr    *outptr;
        glob_cfg        *configptr;

int font_unpmv(flagptr,inptr,outptr,configptr)
        font_flags      *flagptr;
```

```
        font_inptr      *inptr;
        font_outptr     *outptr;
        glob_cfg        *configptr;

int block_move(flagptr,inptr,outptr,configptr)
        blkmv_flags  *flagptr;
        blkmv_inptr  *inptr;
        blkmv_outptr*outptr;
        glob_cfg        *configptr;

int self_test(flagptr,inptr,outptr,configptr)
        test_flags      *flagptr;
        test_inptr      *inptr;
        test_outptr     *outptr;
        glob_cfg        *configptr;

int excep_hdlr(flagptr,inptr,outptr,configptr)
        excep_flags   *flagptr;
        excep_inptr   *inptr;
        excep_outptr *outptr;
        glob_cfg        *configptr;

int inq_conf(flagptr,inptr,outptr,configptr)
        conf_flags      *flagptr;
        conf_inptr      *inptr;
        conf_outptr     *outptr;
        glob_cfg        *configptr;

int set_cm_entry(flagptr,inptr,outptr,configptr)
        conf_flags      *flagptr;
        conf_inptr      *inptr;
        conf_outptr     *outptr;
        glob_cfg        *configptr;

int dma_ctrl(flagptr,inptr,outptr,configptr)
        conf_flags      *flagptr;
        conf_inptr      *inptr;
        conf_outptr     *outptr;
        glob_cfg        *configptr;

int flow_ctrl(flagptr,inptr,outptr,configptr)
        conf_flags      *flagptr;
        conf_inptr      *inptr;
        conf_outptr     *outptr;
        glob_cfg        *configptr;
```

```
int user_timing(flagptr,inptr,outptr,configptr)
        conf_flags    *flagptr;
        conf_inptr    *inptr;
        conf_outptr   *outptr;
        glob_cfg      *configptr;

int process_mgr(flagptr,inptr,outptr,configptr)
        conf_flags    *flagptr;
        conf_inptr    *inptr;
        conf_outptr   *outptr;
        glob_cfg      *configptr;

int sti_util(flagptr,inptr,outptr,configptr)
        conf_flags    *flagptr;
        conf_inptr    *inptr;
        conf_outptr   *outptr;
        glob_cfg      *configptr;
```

Each routine must return a "0" (PASS) if it completed successfully, a "1" (NOT_RDY) if it exited rather than waiting in a busy loop, or a "-1" (FAIL) if it failed during execution.

5.1.2 busy waiting and reentrancy

This type of code can often contain "busy loops" where the code loops at one (or more) spots waiting for a condition to be true (such as the completion of a block move). Some calling routines do not mind busy waiting while others would like to avoid it when possible. To provide for both eventualities, a WAIT flag is passed to each routine to indicate whether busy waiting is acceptable or not. In order to simplify the interface to each routine, all STI ROM routines must be written to check immediately upon entry for potential busy waiting during that call and return if they have been told not to wait. If an STI routine performs the initial test to see if the device is busy and finds that the device is not busy, it can proceed in either of two ways. It can either start and complete all of its activities before it returns, even if this requires busy waiting in the middle of the routine, or it may start and perform part of its activities and then exit rather than busy wait in the middle of the routine. If it is written to exit rather than busy wait in the middle of the routine, it must be written with the following assumptions:

1. The contents of all local variables that the routine was using will be destroyed and must be initialized upon reentry.

2. Before exiting, the routine must change the value of the reent_lvl variable in the global configuration area to a non-zero value that indicates to the routine the point at which it exited and must later reenter.

3. When the routine is next reentered, it will test this reent_lvl variable to determine at what point in the code it must continue from.

4. Upon successful completion of any STI routine, that routine must make sure that the reent_lvl variable has been reset to zero.

5. The calling routine is not allowed to change the value stored in the reent_lvl variable except immediately prior to calling INIT_GRAPH with RESET=1. In this case, the calling routine must set reent_lvl to zero.

It is expected that this method for avoiding busy waiting in the middle of a routine will be rarely used and then usually only when the routine needs to be tuned for performance.

If a routine is told not to wait and detects a busy wait condition and therefore exits, the routine must return NOT_RDY and set reent_lvl to a meaningful value.

Once a calling routine has called an STI routine and been told by the STI routine that it exited without finishing, the calling routine must insure that the graphics device associated with that STI routine is not accessed except by the original STI routine that exited without finishing. This requirement includes preventing any user processes from accessing the device in any manner. Instead it must continue to call the first STI routine with the same structure of input parameters used during the original call until it is told by that STI routine that it successfully completed. If an STI routine does return without completing, that routine is allowed to use the memory pointed to by the save_addr pointer in glob_conf for saving some state information between repeated calls. The calling routine is, of course, allowed to do other non-STI related activities between these repeated calls.

The size of the area provided by the save_addr pointer in glob_conf is given in the device dependent (DD) data section of the STI ROM as a two byte value called REENTSIZE Storage.

The INIT_GRAPH graph routine must handle a special situation of busy waiting. If this routine is told to do a hard reset it must ignore the state of the wait flag completely until after the reset is done. This is necessary since the graphics device could be "hung" and if it waited before resetting the device would never get reset.

5.1.3 memory usage

The following limits apply to how much memory can be used by the ROM routines. These limits are generally required due to Boot ROM constraints.

- Program space - 10K bytes. No single (e.g., SELF_TEST) STI ROM routine may require more than 10K bytes of RAM for execution. This limit is set by the amount of space that the Boot ROM can allocate for additional programs.

- Configuration Data Space - 100 bytes. The data structure for storing the global configuration data (see above) must not exceed 100 bytes. This limit does not include any space needed for reentrant data storage (see the section on Busy Waiting).

- Local variable storage on stack - 5k bytes. The data space required by any ROM routine for local storage must not exceed 5k bytes.

No limit is put on the size of the data space that can be required for state data by the STATE_MGMT routine as the boot ROM is not expected to use this feature.

Those STI ROM code developers interested in the performance of their graphics devices should also try to insure that the combined size of the FONT UNPMV and BLOCK_MOVE routines does not exceed 10k bytes. If the combined size of these routines is kept to less than 10k bytes, then the boot ROM will be able to keep both routines in memory together. Otherwise, the swapping of routines may significantly affect console performance during booting. This performance consideration is only a potential problem during booting and does not affect kernel performance at all.

5.1.4 error returns

All the STI routines have the capability to return an error number should the routine fail during execution. The device dependent error numbers for each routine must be documented in each graphics device's ERS. Device dependent error numbers must have values between 0x100 and 0x1ff.

The device independent error numbers are documented in the Appendix of this document. Device independent error numbers must have values between 0x00 and 0xff.

5.1.5 end address

The end of each routine in ROM can be found by using the pointer to the start of the next routine. The last byte of a routine is simply found by subtracting 0x04 from the pointer to the start of the following routine. The code for each routine must be organized in the same order in which the pointers are supplied in the DD section of the ROM. In addition, the routines must be located in ROM such that the only unused space between them is what is necessary to insure that the next routine is aligned on a 32 bit boundary. The end of the last routine is found by subtracting 0x04 from the "End Address" pointer supplied in the DD section.

Note that all listed routines are required and must have entry points even if the routine only contains a RETURN.

5.1.6 Global Config

As mentioned, the global configuration structure is jointly managed by the caller of the STI routines and the init_graph() STI routine.

Here is the current definition of the global config structure. You can see based on the glob_cfg_ext structure that the base glob_cfg structure has been extended over the years.

```
typedef struct {
   int32_t    text_planes;              /* number of planes used for text */
   int16_t    onscreen_x;               /* screen width in pixels */
   int16_t    onscreen_y;               /* screen height in pixels */
   int16_t    offscreen_x;              /* offscreen width in pixels */
   int16_t    offscreen_y;              /* offscreen height in pixels */
   int16_t    total_x;                  /* frame buffer width in pixels */
   int16_t    total_y;                  /* frame buffer height in pixels */
   int32_t    *region_ptrs[REGION_MAX];   /* region pointers */
   int32_t    reent_lvl;                /* storage for reentry level value */
   int32_t    *save_addr;               /* where to save or restore reentrant state */
   glob_cfg_ext *ext_ptr;               /* pointer to extended glob_cfg data structure */
} glob_cfg;
```

The text_planes value is set by the init_graph() routine and is an indicator to all other STI routines of the number of planes setup for the console.

The onscreen_x, onscreen_y, offscreen_x, offscreen_y, total_x and total_y values are setup by the init_graph() routine. These values indicate the current resolution in pixels of the device as well as the total framebuffer space available for device configuration.

The reent_lvl and save_addr were described earlier. This is the mechanism that the STI routines use to be re-entrant if they perform busy waiting.

The region_ptrs array is the one part that is managed by the caller, whether it is the boot ROM or the kernel. This is an array of the various spaces that have been mapped by the caller and are based on the REGION LIST in the DD struct.

The ext_ptr points to the extended glob_cfg_ext structure.

```
typedef struct {
   uint8_t    curr_mon;                 /* current monitor configured */
   uint8_t    friendly_boot;            /* in friendly boot mode */
   int16_t    power;                    /* power calculation */
```

```
   int32_t    freq_ref;                /* frequency refrence */
   int32_t    *sti_mem_addr;         /* pointer to global sti memory */
   int32_t    *future_ptr;             /* pointer to future data */
} glob_cfg_ext;
```

The curr_mon value indicates the current selected monitor for a device that supports multiple monitors. The caller can use this to identify which monitor is current and if the one desired is different then init_graph() is called, which has an inptr element that identifies the new monitor to select.

The friendly_boot value indicates that the STI is using the early console capability of the device. This was common in 712/715 and 725 based products but are not used in current generation products.

The power value is a maximum power dissipated (Watts) for the graphics device. This value is commonly used by the firmware to set the speed of the fans inside the SPU.

The freq_ref value is provided for the STI when the graphics hardware clocks are based on the bus clock. This was common for some of the older graphics architectures but is no longer used.

The sti_mem_addr field is a pointer to global memory that is guaranteed to be modified only by STI code and is consistent between calls to STI. The address points to an area of memory that has been set aside by the caller of STI and is of size STI_MEM_REQUEST as specified in the DD Info part of the STI ROM.

5.2 init graph

Definition:

```
int32_t init_graph(&init_flags,&init_inptr,&init_outptr,&glob_cfg)
```

Flagptr structure :

```
typedef struct {
   uint32_t   wait                  : 1;   /* should routine idle wait or not */
   uint32_t   reset                 : 1;   /* hard reset the device? */
   uint32_t   text                  : 1;   /* turn on text display planes? */
   uint32_t   nontext               : 1;   /* turn on non-text display planes? */
   uint32_t   clear                 : 1;   /* clear text display planes? */
   uint32_t   cmap_blk              : 1;   /* non-text planes cmap black? */
   uint32_t   enable_be_timer       : 1;   /* enable bus error timer */
   uint32_t   enable_be_int   : 1;   /* enable bus error timer interrupt */
   uint32_t   no_chg_tx             : 1;   /* don't change text settings */
   uint32_t   no_chg_ntx      : 1;   /* don't change non-text settings */
```

```
    uint32_t    no_chg_bet      : 1;   /* don't change berr timer settings */
    uint32_t    no_chg_bei      : 1;   /* don't change berr int settings */
    uint32_t    init_cmap_tx    : 1;   /* initialize cmap for text planes */
    uint32_t    cmt_chg                : 1;   /* change current monitor type */
    uint32_t    retain_ie              : 1;   /* don't allow reset to clear int
                                                 enables */
    uint32_t    pad                    : 17;  /* pad to word boundary */
    int32_t     *future_ptr;           /* pointer to future data */
} init_flags;
```

Inptr and Inptr Extended Structures:

```
typedef struct {
    int32_t     text_planes;       /* number of planes to use for text */
    init_inptr_ext *ext_ptr;               /* pointer to extended init_graph inptr data
structure*/
} init_inptr;

typedef struct {
    uint8_t     config_mon_type;       /* configure to monitor type */
    uint8_t     pad[1];                /* pad to word boundary */
    uint16_t    inflight_data;     /* inflight data possible on PCI */
    int32_t     *future_ptr;       /* pointer to future data */
} init_inptr_ext;
```

Outptr Structure :

```
typedef struct {
    int32_t     errno;                 /* error number on failure */
    int32_t     text_planes;       /* number of planes used for text */
    int32_t     *future_ptr;       /* pointer to future data */
} init_outptr;
```

At power-up, the graphics device needs to be initialized with video timing constants, color map values, and other conditions needed to enable the display to support text output. The ROM routines may support from two to eight color of text, depending on the capabilities of the graphics device. The values of these colors are:

- 0-Black
- 1-White
- 2-Red
- 3-Yellow

- 4-Green
- 5-Cyan
- 6-Blue
- 7-Magenta

To allow compatibility with legacy products warm reboot approach, this routine must not, by default, clear the onscreen portion of the display. Instead, a flag is available to allow the clear to be done if appropriate.

At boot time, the boot ROM must execute the initialization code followed by the selftest code on each graphics device that it finds.

One type of reset is defined, hard reset. The proper setting of the available flags in this routine can perform the operation known as “soft” reset. A separate soft reset capability is therefore not needed.

If hard reset is requested, the wait flag must be ignored until the reset has been completed.

Hard reset should assume that the device has ended up in the worst state possible and therefore everything must be reset and re-initialized. In general, hard reset must first duplicate the action that occurs during an actual power on reset. In most cases, having the STI routine actually assert the reset line to the graphics devices easily does this. Following this, several other actions must be taken.

It should be assumed that the text and non-text planes contain garbage and display of these planes should be disabled unless the proper flags are set to enable the display of the text or non-text planes. In addition, the hard reset routine should allow action to be taken to insure that if the non-text planes become enabled, that nothing will be displayed. Therefore a flag is provided to allow this routine to be told to set all the color map entries in the non-text planes to black, thereby effectively blanking the display of the non-text planes. The state of this flag must be ignored unless a hard reset is also requested at the same time. In this manner, the HP-UX needs of not having anything displayed by the non-text planes after hard reset can be met, while the DOMAIN needs of having the contents of the non-text planes displayed can also be met.

In addition, hard reset must set all constant registers used by the STI routines to usable values. It must disable blinking if available, initialize the text plane(s) color map, and set overlay color black to be transparent and disable the z buffer, if available.

Four flags are also available to allow the calling routines to request that certain parts of the graphics device's state do not change. Calling routines are able, by using the proper

setting of these flags and other flags, to individually enable or disable each function available in INIT_GRAPH.

This routine must be passed a pointer to the following flags, via the init_flags pointer:

- A flag to indicate whether the ROM routine may wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate whether (RESET=1) or not (RESET=0) the device should be hard reset.

- A flag to indicate whether the planes used by the ROM code for text display should be turned on (TEXT=1) or off (TEXT=0).

- A flag to indicate whether the planes not used by the ROM code text display should be turned on (NONTEXT=1) or off (NONTEXT=0).

- A flag to indicate whether the planes used by the ROM code for text display should be cleared to black (CLEAR=1) or not (CLEAR=0). The full extent of the text planes is cleared by this operation. (This clear is intended only for use immediately after power up. For all other cases, the block move routine should be used to perform the clear.)

- A flag to indicate whether all the color map entries used by the non-text planes should be cleared to black (CMAP_BLK=1) or not (CMAP_BLK=0). This flag is ignored unless a hard reset is also requested.

- A flag to indicate whether the device must catch possible bus errors with its internal timer, set a status bit and complete the cycle (ENABLE_BE_TIMER=1) or whether the device should let an actual bus error occur on the bus (ENABLE_BE_TIMER=0). If this functionality is not supported on the device then this flag should not perform any action. All graphics devices should provide this functionality. The effects of not providing this functionality are that accesses to addresses that are not decoded will result in an HPMC on PA-RISC platforms.

- A flag to indicate whether the device should generate an interrupt if it has caught a bus error with its internal timer (ENABLE_BE_INT=1) or not (ENABLE_BE_INT=0). If this functionality is not supported on this device, then this flag should not perform any action and a device independent error (NO_BE_INTR) should be returned (see Appendix).

- A flag to indicate whether the state of the planes used by the ROM code for text display should be changed (NO_CHG_TX=0) or not (NO_CHG_TX=1). If this flag is one, then the TEXT flag is ignored and the display of the text

planes is left unchanged. If this flag is zero, then the display of the text is enabled or disable based on the setting of the TEXT flag.

- A flag to indicate whether the state of the planes not used by the ROM code for text display should be changed (NO_CHG_NTX=0) or not (NO_CHG_NTX=1). If this flag is one, then the NONTEXT flag is ignored and the display of the non-text planes is left unchanged. If this flag is zero, then the display of the non-text is enabled or disable based on the setting of the NONTEXT flag.

- A flag to indicate whether the state of the internal bus error timer on the device should be changed (NO_CHG_BET=0) or not (NO_CHG_BET=1). If this flag is one, then the ENABLE_BE_TIMER flag is ignored and the state of the bus error timer is left unchanged. If this flag is zero, then the bus error timer is enabled or disable based on the setting of the ENABLE_BE_TIMER flag.

- A flag to indicate whether the state of the internal bus error interrupt on the device should be changed (NO_CHG_BEI=0) or not (NO_CHG_BEI=1). If this flag is one, then the ENABLE_BE_INT flag is ignored and the state of the bus error interrupt is left unchanged. If this flag is zero, then the bus error interrupt is enabled or disable based on the setting of the ENABLE_BE_INT flag.

- A flag to indicate whether the text planes color map should be initialized (INIT_CMAP_TX=1) or not (INIT_CMAP_TX=0).

- A flag to indicate whether the call to init_graph includes a request to change the current monitor type (CMT_CHG=1) or not (CMT_CHG=0).

- A flag to indicate the caller requires the current interrupt enable settings be saved (RETAIN_IE=1) or not (RETAIN_IE=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the init_inptr pointer:

- The number of planes (1,2, or 3) which the calling routine would like to have used for text display by the STI routines.

- A pointer (ext_ptr) to the init_inptr_ext structure.

This routine must be passed a pointer to a structure containing the following, via the ext_ptr in the init_inptr pointer:

- The monitor type to configure, CONFIG_MON_TYPE.

- The amount of inflight data that may be issued by the CPU but not yet seen by the graphics card (INFLIGHT_DATA)

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure. Via the init_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- The number of planes (text_planes) which will be used by the STI routines for display of text. This same value must also be returned in the glob_conf structure.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine also uses the "glob_conf" structure that contains the following values. This routine or the calling routine may set these values. The primary purpose of the global configuration structure is to provide an area of global data that can be shared among the various STI routines. Some of the values, such as display width and height, in this global area may be constants for certain graphics devices, but also may vary in graphics devices that support multiple monitor configurations.

It is important to remember that this glob_conf area must be set up before any of the STI routines can successfully run. Therefore, INIT_GRAPH must always be the first routine run when starting to use each graphics device, independent of whether that device is the console or not. A good approach would be for the kernel to set up the glob_conf for each device as it is doing the initial scan of its I/O space to see what devices exist.

The calling process (e.g., kernel or boot ROM) must perform the following steps to insure that the glob_conf area is set up properly. First, in the glob_conf structure, it must enter the starting addresses (virtual or physical, as appropriate) for the device's regions into the region array and set the reent_lvl value to zero. Then it must call the INIT_GRAPH routine with RESET=1. After INIT_GRAPH returns, the glob_conf structure will be properly set up.

- The number of planes (text_planes) which will be used by the STI routines for display of text.

- Onscreen display width (onscreen_x), in pixels. (e.g., 1024).

- Onscreen display height (onscreen_y), in pixels.
- Offscreen display width (offscreen_x), in pixels.
- Offscreen display height (offscreen_y), in pixels.
- Total display width (total_x), in pixels.
- Total display height (total_y), in pixels.
- An array of size eight which will contain the actual virtual addresses to which the memory regions in the region list have been mapped. The calling routines (kernel, boot ROM or other) must map in all the regions in the region list into this array prior to calling any STI routines. This region list must always be used for accesses to the device. Unused entries in this array must be null.
- An integer (reent_lvl) which can used by the STI routines to keep track of where within the routine it exited, if it have been told to not busy wait via the WAIT flag. See the section on "Busy Waiting" for more discussion on the use of this value.
- A pointer (save_addr) to the data space that this routine must use for saving and restoring reentrant level state. See the section on "Busy Waiting" for more information on using this space. This pointer must be null if this space is not provided.

The extension to the global config structure will also be initialized, glob_cfg_ext:

- The current monitor in effect (CURR_MON).
- The current state of early console support (FRIENDLY_BOOT).
- The maximum amount of power used by the card in the current configuration.
- The caller for graphics products that base their internal operations on the speed of the bus frequency must set the frequency reference. This has only been used by 712/715 and 725 based products.
- Likewise, the caller must set the sti_mem_addr before calling init_graph. This value is used to pass global data among the various STI routines.
- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

The actual number of text planes that will be used by the STI routines is contained in the text_planes variable in glob_conf. The final value that is returned here is determined as follows. The actual number of text planes which will be used is the smaller of 1) the maximum number of text planes which this device can support, and 2) the number of text_planes which were requested (1-3) via the input structure to this routine. For example, if the device is a monochrome device (one text plane max), and three text planes are requested as part of the input to this routine, the text_planes value in glob_conf will contain a one.

The x,y pixel locations (onscreen, offscreen and total) are assumed to be referenced to a 0,0 location in the upper left hand corner of the onscreen portion of the display. The x-axis is assumed to have increasing values as we move from left to right across the screen and the y-axis is assumed to have increasing values as we move from top to bottom of the screen.

If there is no offscreen area available, then the onscreen width and height will be equal to the offscreen width and height. If there is no addition framebuffer beyond the offscreen area, then the offscreen width and height will be equal to the total width and height.

INIT_GRAPH returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.3 state management

Definition:

```
int32_t state_mgmt(&state_flags,&state_inptr,&state_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t   wait                  : 1;  /* should routine idle wait or not */
   uint32_t   save                  : 1;  /* save (1) or restore (0) state */
   uint32_t   res_disp              : 1;  /* restore all display planes */
   uint32_t   pad                   : 29; /* pad to word boundary */
   int32_t    *future_ptr;       /* pointer to future data */
} state_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t    *save_addr;             /* where to save or restore state */
   int32_t    *future_ptr;       /* pointer to future data */
```

```
} state_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;          /* error number on failure */
   int32_t    *future_ptr;    /* pointer to future data */
} state_outptr;
```

This routine will save or restore the state of the graphics device when called. This is needed to allow the ITE to use the device at the same time that other graphics processes may be using it.

If a state save is being done, this routine must first insure that the graphics device has completed any pending operations and is in a state that can be saved and later restored.

If a state restore is being done, this routine must first insure that the graphics device has completed any operations that were started after the state save (for instance, by the ITE) and that the device is now in a quiescent state.

This routines should always save the display state of the text and non-text planes (e.g., display enables, etc) but should only restore this state if told to via the appropriate flag.

This routine must be passed a pointer to the following flags, via the state_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate whether a state save (SAVE=1) or a state restore (SAVE=0) should be done.

- A flag to indicate whether the display state of the text and non-text planes should be restored (RES_DISP=1) or not (RES_DISP=0) when a state restore is done. This does not include the contents of the text and non-text planes.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the state_inptr pointer:

- A word aligned pointer (save_addr) to the data space that this routine must use for saving or restoring state.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the state_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

STATE_MGMT returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.4 font unpack/move

Definition:

```
int32_t font_unpmv(&font_flags,&font_inptr,&font_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                    : 1;  /* should routine idle wait or not */
   uint32_t    non_text                : 1;  /* font unpack/move in non_text
                                                planes =1,text =0 */
   uint32_t    pad                     : 30; /* pad to word boundary */
   int32_t     *future_ptr;        /* pointer to future data */
} font_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t     *font_start_addr;          /* address of font start */
   int16_t     index;              /* index into font table of character */
   uint8_t     fg_color;           /* foreground color of character */
   uint8_t     bg_color;           /* background color of character */
   int16_t     dest_x;                    /* X location of character upper left */
   int16_t     dest_y;             /* Y location of character upper left */
   int32_t     *future_ptr;        /* pointer to future data */
} font_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;          /* error number on failure */
   int32_t    *future_ptr;    /* pointer to future data */
} font_outptr;
```

This routine will allow a character to be moved from the font storage area in the STI ROM (or another font storage area of the same format), to a given x,y location in the text planes. It will unpack that character into a suitable format for displaying while it is performing the font move.

This routine can either be used to put characters directly onto the onscreen portion of the display or to put them into offscreen memory. If they are put into offscreen memory in the neutral format (see below), then the BLOCK_MOVE routine can be used to quickly move them from offscreen storage to onscreen display as needed.

The font storage area must be configured as described later in this document. In particular, it can only be set up to use the least significant byte of each word. This is required even if the calling routine (e.g., the kernel) has provided an alternate font in system RAM.

This routine must be passed a pointer to the following flags, via the font_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate that the character should be unpacked into the non-text planes (NON_TEXT=1) or the text planes (NON_TEXT=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the font_inptr pointer:

- The address (font_start_addr) of the start of the font storage area from which to get the character. This address must be a full 32-bit address (virtual or physical, as required) which will let the STI code access the font table directly. (See the "Font Storage Format" section for more information on how to use this address.)

- The index (index) into the font table of the character to put on the screen. This is the ASCII value of the character to be displayed.

- The foreground color (fg_color) to be used (0-7)

- The background color (bg_color) to be used (0-7).

- The x pixel location (dest_x) of the upper left hand corner of the rectangle it will be moving the character to.

- The y pixel location (dest_y) of the upper left hand corner of the rectangle it will be moving the character to.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the font_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

Inverse video can be accomplished by swapping the values of the foreground and background colors.

The address of the font storage area must be an absolute address that includes valid values for the upper six bits of the address. This is required since the font storage area may exist outside of the graphics memory space of the card.

Note that the range of values for the input parameters for foreground and background colors is dependent on the number of planes being used for text. For instance, the only valid parameter values for a single plane system are 0 and 1. Trying to use any other values may results in an error being returned by the routine being accessed.

FONT_UNP/MV returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.5 block move

Definition:

```
int32_t block_move(&blkmv_flags,&blkmv_inptr,&blkmv_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                    : 1;  /* should routine idle wait or not */
```

```
    uint32_t    color                   : 1;  /* change color during move? */
    uint32_t    clear                   : 1;  /* clear during move? */
    uint32_t    non_text                : 1;  /* block move in non_text planes
                                                 =1,text =0 */
    uint32_t    pad                     : 28; /* pad to word boundary */
    int32_t     *future_ptr;        /* pointer to future data */
} blkmv_flags;
```

Inptr Structure:

```
typedef struct {
    uint8_t     fg_color;               /* foreground color after move */
    uint8_t     bg_color;               /* background color after move */
    int16_t     src_x;                  /* source upper left pixel x location */
    int16_t     src_y;                  /* source upper left pixel y location */
    int16_t     dest_x;                 /* dest upper left pixel x location */
    int16_t     dest_y;                 /* dest upper left pixel y location */
    int16_t     width;                  /* block width in pixels */
    int16_t     height;                 /* block height in pixels */
    int32_t     *future_ptr;            /* pointer to future data */
} blkmv_inptr;
```

Outptr Structure:

```
typedef struct {
    int32_t     errno;                  /* error number on failure */
    int32_t     *future_ptr;            /* pointer to future data */
} blkmv_outptr;
```

This routine will allow a variable size block to be moved from one x,y location in the text planes to another x,y location in the text planes. Unless the color flag is set, this routine must insure that these attributes of the block being moved are unchanged by the move.

This routine may provide improved performance when used with the FONT_UNP/MV routine in the following manner. The FONT_UNP/MV would be used to move the entire font into offscreen memory and store it in the "neutral" format. The BLOCK_MOVE routine can then quickly and easily move characters from offscreen to onscreen as needed.

This routine will also perform a clear of a variable size block in the text planes if the clear flag is set.

This routine must be written such that the "neutral" form for storing characters on the screen is the same as for writing a character in the color represented by all "1's" on a black background on the screen. That is, neutral format is as follows:

- 3 text planes - magenta on black
- 2 text planes - yellow on black
- 1 text planes - white on black

The "neutral" form is that form such that if color or inverse video is applied, then the expected result is obtained. For example, if the application wanted to store the font in offscreen storage and then move these characters onto onscreen display as inverse video, and two text planes were being used, the offscreen characters must be stored as yellow on black characters. This block move routine must then be written to perform the proper translation on these neutral characters if the color is changed. Applying colors to a block that is not stored in neutral format will yield indeterminate results.

Performing a clear will set the destination rectangle to the background color and will ignore the foreground color and source rectangle. The dest_x, dest_y, width, and height are used to specify the region to be cleared.

This routine must be passed a pointer to the following flags, via the blkmv_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).
- A flag to indicate whether the color should change (COLOR=1) or not (COLOR=0) during the move.
- A flag to indicate whether to perform a clear (CLEAR=1) or a move (CLEAR=0).
- A flag to indicate that the block move should be performed in the non-text planes (NON_TEXT=1) or the text planes (NON_TEXT=0).
- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the blkmv_inptr pointer:

- The foreground color (fg_color) of the block after the move.
- The background color (bg_color) of the block after the move
- The x pixel location (src_x) of the upper left corner it will be moving the block from.

- The y pixel location (src_y) of the upper left corner it will be moving the block from.

- The x pixel location (dest_x) of the upper left corner it will be moving the block to.

- The y pixel location (dest_y) of the upper left corner it will be moving the block to

- The width (width) of the block to be moved, in pixels.

- The height (height) of the block to be moved, in pixels.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the blkmv_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

BLOCK_MOVE returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.6 self test

Definition:

```
int32_t self_test(&test_flags,&test_inptr,&test_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                : 1;  /* should routine idle wait or not */
   uint32_t    ext_test            : 1;  /* perform extended self test */
   uint32_t    pad                 : 30; /* pad to word boundary */
   int32_t     *future_ptr;        /* pointer to future data */
} test_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t    *future_ptr;       /* pointer to future data */
} test_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;             /* error number on failure */
   int32_t    result;                    /* result of the self test */
   int32_t    *future_ptr;       /* pointer to future data */
} test_outptr;
```

Also included in the ROM is a self-test routine. This routine should include a normal self test and may also include an extended self test. The normal self test should attempt to test as much of the graphics hardware functionality as is needed to display test on the graphics device while the extended self test should attempt to test the entire graphics device. The normal self test should be designed to take no longer than four seconds to complete its test of the device.

This routine must be passed a pointer to the following flags, via the test_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate whether the routine should perform its extended self test (EXT_TEST=1) or its normal self test (EXT_TEST=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the test_inptr pointer:

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the test_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A failure result (result) which contains device dependent information regarding what failed self test. Refer to the service information or device's

ERS for the specific device to determine the meaning of the contents of this result.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

A self-test routine should be written to enable failures to be isolated to a field replaceable unit (FRU). In general, a FRU is a printed circuit board or a set of PCB's which share significant commonality. The test of each FRU should be as extensive as possible but any failure information returned must allow the defective FRU(s) to be easily identified. A result value of zero indicates no failure; while a non-zero value indicates a failure. If a failure is indicated, the specific value returned in the result must contain device dependent information about the type of failure that occurred. The calling routine must attempt to display the value of this returned result for use in servicing the device. Since a failure of this device could mean that the system will not have a graphics device with which to communicate failure data, the graphics device designer also needs to provide a means of making this detailed failure information available for service. This has typically been done by providing a set of LED's in the graphics device that the STI routine must set to also show the failure status.

SELF_TEST returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.7 exception handler

Definition:

int32_t excep_hdlr(&excep_flags,&excep_inptr,&excep_outptr,&glob_cfg)

Flagptr Structure:

```
typedef struct {
   uint32_t   wait              : 1;   /* should routine idle wait or not */
   uint32_t   clr_int           : 1;   /* should routine clr int or not */
   uint32_t   clr_be            : 1;   /* should routine clr be stat or not */
   uint32_t   save_int          : 1;   /* should int state be saved or not */
   uint32_t   restore_int       : 1;   /* should int state be restored or
                                                        not*/
   uint32_t   write_eim                : 1;   /* write eim with inptr_ext  */
   uint32_t   read_eim                 : 1;   /* read eim to outptr_ext */
   uint32_t   global_int_enable        : 1;
   uint32_t   no_chg_gie               : 1;
   uint32_t   write_int_enb_mask       : 1;
   uint32_t   read_int_enb_mask        : 1;
```

```
    uint32_t    begin_int_cycle          : 1;
    uint32_t    end_int_cycle            : 1;
    uint32_t    retain_ie          : 1;   /* don't allow reset to clear int
                                                          enables */
    uint32_t    pad                : 18;  /* pad to word boundary */
    int32_t     *future_ptr;              /* pointer to future data */
} excep_flags;
```

Inptr and Inptr Extended Structures:

```
typedef struct {
    int32_t     *save_addr;               /* where to save or restore int state */
    excep_inptr_ext  *ext_ptr;
} excep_inptr;

typedef struct {
    uint32_t    eim_addr;
    uint32_t    eim_data;
    uint32_t    int_enable_mask;        /* for setting int enables */
    uint32_t    int_clear_mask;         /* for clearing pending ints */
    int32_t     *future_ptr;
} excep_inptr_ext;
```

Outptr and Outptr Extended Structures:

```
typedef struct {
    int32_t     errno;              /* error number on failure */
    uint32_t    be                        : 1;   /* was be intercepted or not */
    uint32_t    int_pend                  : 1;   /* is there an existing int or not */
    uint32_t    ints_enabled              : 1;   /* is global enable set ? */
    uint32_t    pad                       : 29;  /* pad to word boundary */
    excep_outptr_ext    *ext_ptr;
} excep_outptr;

typedef struct {
    uint32_t    eim_addr;
    uint32_t    eim_data;
    uint32_t    int_enable_mask;        /* for readback of enables */
    uint32_t    int_state;
    int32_t     *future_ptr;
} excep_outptr_ext;
```

Also included in the ROM is an exception handler routine. This routine handles two types of exceptions.

The first is the special bus error trapping functionality that is now required on all graphics devices. This functionality allows the graphics device to be set up to complete a pending cycle and set a status bit, and in some cases also generate an interrupt, rather than allowing a bus timeout. Refer to the specific bus specification for more details on this functionality.

The second is handling for all the various types of interrupts which any particular graphics device could generate.

This routine provides the capability to test for and/or clear both of these types of exceptions. In addition, it provides the capability to save the state, enabled or disabled, of the device's interrupt masks and mask, disable, all the interrupts, or to restore the state of the interrupt masks.

There are two basic methods for generating interrupts, one that is not bus specific is called EIM and is considered a directed interrupt. The other is specific to the PCI bus and uses the PCI INT lines, and is considered a polled interrupt.

The directed interrupt, using eim_addr and eim_data directs the interrupt to a specific processor by writing the eim_addr with eim_data. When an interrupt occurs, the hardware will write the data contained in the eim_data register to the bus address contained in the eim_addr register. This allows the processor to know exactly which piece of hardware generated the interrupt.

The PCI mechanism requires the hardware to pull the PCI INT line when an interrupt occurs. Since other cards on the PCI bus can share the PCI INT line, the kernel must poll every possible interrupt source on the PCI bus to determine who caused the interrupt.

This routine must be passed a pointer to the following flags, via the excep_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate whether the ROM routine should clear an existing interrupt (CLR_INT=1) or not (CLR_INT=0). All device interrupts, with the exception of an interrupt caused by the graphics device intercepting a bus error, should be cleared. (see below).

- A flag to indicate whether the ROM routine should clear the bus error status bit (CLR_BE=1) or not (CLR_BE=0). In addition to clearing the bus error status bit, this flag should also clear an interrupt generated by the bus error timer, if such an interrupt is supported and asserted on this graphics device.

- A flag to indicate whether the state (enabled or disabled) of the device's interrupt masks should be saved and that all interrupts in the device should be

masked (disabled) (SAVE_INT=1) or not (SAVE_INT=0). This must be done for all interrupts, including the interrupt that may be supported as part of the bus error timeout circuitry.

- A flag to indicate whether the saved state of the device's interrupt masks should be restored (RESTORE_INT=1) or not (RESTORE_INT=0). This operation must restore the device's interrupt masks to the state that was saved by the SAVE_INT flag. This must be done for all interrupts, including the interrupt that may be supported as part of the bus error timeout circuitry.

- A flag to indicate that a write to the EIM register is desired (WRITE_EIM=1). The data that is written is passed in through the extended intptr. See the eim_addr and eim_data entries in the excep_inptr_ext structure.

- A flag to indicate that a read of the EIM register is desired (READ_EIM=1). The result is returned via the extended outptr. See the eim_addr and eim_data entries in the excep_outptr_ext structure.

- A flag to indicate that all interrupts should be enabled (GLOBAL_INT_ENABLE=1) or that all interrupts should be disabled (GLOABAL_INT_ENABLE=0). This flag is only valid if the NO_CHG_GIE is clear, allowing changes the enabled state.

- A flag to indicate that no change to the global interrupt should be performed (NO_CHG_GIE=1) or that the value in the GLOABAL_INT_ENABLE should be decoded (NO_CHG_GIE=0).

- A flag to indicate that a write of the interrupt enable mask is desired (WRITE_INT_ENB_MASK=1) or not desired (WRITE_INT_ENB_MASK=0). This flag indicates that the data in the int_enable_mask entry of the extended inptr structure is valid.

- A flag to indicate that a read of the interrupt enable mask is desired (READ_INT_ENB_MASK=1) or not desired (READ_INT_ENB_MASK=0). The data is returned via the int_enable_mask element of the extended outptr structure.

- A flag to indicate that an interrupt cycle should be started (BEGIN_INT_CYCLE=1) or not (BEGIN_INT_CYCLE=0).

- A flag to indicate that an interrupt cycle should be ended (END_INT_CYCLE=1) or not (END_INT_CYCLE=0).

- A flag to indicate that a reset should not clear the current state of the interrupt enables (RETAIN_IE=1) or not (RETAIN_IE=0). There is not a specific

"reset" flag, it is assumed that if the CLR_BE flag is set, the card must be reset.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the excep_intpr pointer:

- A word aligned pointer (save_addr) which should point to a single 32 bit location for saving or restoring the state of the interrupt masks. The contents of this location must be maintained by the kernel to later restore the state of the interrupt masks. The flags to control the save and restore of the interrupt state are from the flagptr and are save_int and restore_int flags. The save_addr, save_int and restore_int are used primarily with older graphics devices. This was before the kernel was architected to care about the type of interrupt, other than the bus error interrupt. This mechanism assumed that the interrupt handler, on a call to save_int, would save the current interrupt state and disable all interrupts. And on a call of restore_int it would restore the interrupt enables and re-enable all interrupts. This was only used on the Visualize-48 product for texture map interrupts.

- A pointer (ext_ptr) to the extended intptr pointer structure.

The routine must be passed a pointer to the extended inptr structure, via the ext_ptr in the excep_inptr:

- The address of the CPU EIM register (EIM_ADDR).

- The data to write to the CPU EIM register (EIM_DATA).

- The setting of the interrupt enable mask (INT_ENABLE_MASK). This mask is defined by the int_desc structure in sti.h

- The setting of the interrupt clear mask (INT_CLEAR_MASK). This mask is defined by the int_desc structure in sti.h

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the excep_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A flag to indicate whether a potential bus timeout has been intercepted by this device (BE=1) or not (BE=0) rather than allowed to cause an actual bus error. If the bus error interrupt functionality has been enabled for this device, a value of BE=1 will also indicate that this device is interrupting due to an intercepted bus error.

- A flag to indicate whether this device has interrupted (because of a device dependent interrupt) and not yet been cleared (INT_PEND=1) or not (INT_PEND=0). This flag will not indicate interrupts due to bus errors.

- A flag to indicate that the global interrupt enable has been set on exit from the excep_hdlr routine (INTS_ENABLED=1) or not (INTS_ENABLED=0).

- A pointer (ext_ptr) to the extended outptr structure is passed by the caller.

This routine must be passed the extended ptr to the outpr_extended structure and must return the following via the outptr_extended structure:

- The current value of the eim address register.

- The current value of the eim data register.

- The current value of the interrupt enable mask This mask is defined by the int_desc structure in sti.h

- The current value of the interrupt state. This mask is defined by the int_desc structure in sti.h/

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

The BEGIN_INT_CYCLE and END_INT_CYCLE flags are used to control the hardware such that it will perform an atomic operation of saving the current interrupt state and disabling all further interrupts. This will guarantee that no interrupts are missed. The kernel will first call the excep_hdlr with the BEGIN_INT_CYCLE flag set. It will then make multiple calls to excep_hdlr to read the state of the interrupt mask, clearing various interrupts and then resetting the interrupt mask. Once the kernel interrupt cycle is complete it will make a final call to excep_hdlr with END_INT_CYCLE flag set, causing the hardware to re-enable interrupt processing.

The current definition of the int_desc structure is:

```
typedef struct {
   uint32_t   bet      : 1;     /* bus error timer */
   uint32_t   hw       : 1;     /* high water */
   uint32_t   lw       : 1;     /* low water */
```

```
    uint32_t   tm       : 1;    /* texture map */
    uint32_t   vb       : 1;    /* vertical blank */
    uint32_t   udc      : 1;    /* un-buffered dma complete */
    uint32_t   bdc      : 1;    /* buffered dma complete */
    uint32_t   udpc     : 1;    /* un-buffered priv dma complete */
    uint32_t   bdpc     : 1;    /* buffered priv dma complete */
    uint32_t   pad      : 23;
} int_desc;
```

Most of these should be obvious from the comments. There are both privileged and non-privileged as well as buffered and un-buffered paths for DMA, thus 4 different interrupts.

The kernel can use the 64 graphics id from the DD structure, see section 4.2.2.5, to determine which card it is working with, and knowing which card, knows what set of interrupts are supported.

A general method would be to set the INT_ENABLE_MASK, in the excep_inptr_ext, to –1 and set the WRITE_INT_ENB_MASK flag. Then read back the INT_ENABLE_MASK from the excep_outptr_ext structure, while setting the READ_INT_ENB_MASK flag. The excep_hdlr routine will only return with the bits set for the interrupts that are supported for this particular card. Once this is complete, write the INT_CLEAR_MASK with –1 to clear all supported interrupts. As long as the GLOBAL_INT_ENABLE flag is not set there is no danger of inadvertently causing an unwanted interrupt.

If both the SAVE_INT and RESTORE_INT flags are set, no action to save, restore, or change the state of the masks should be taken.

If the EXCEP_HDLR routine is unable to clear the bus error when asked to, it should return an UNEXPECTED_BE error code to the calling routine.

All devices may not support interrupts but all must provide an EXCEP_HDLR entry point even if the routine only does a return when called.

EXCEP_HDLR returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.8 inquire config

Definition:

```
int32_t inq_conf(&conf_flags,&conf_inptr,&conf_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                    : 1;  /* should routine idle wait or not */
   uint32_t    pad                     : 31; /* pad to word boundary */
   int32_t     *future_ptr;        /* pointer to future data */
} conf_flags;
```

Inptr and Inptr Extended Structures:

```
typedef struct {
   int32_t     *future_ptr;        /* pointer to future data */
} conf_inptr;

typedef struct {
   uint32_t    crt_config[3];
   uint32_t    crt_hdw[3];
   int32_t     *future_ptr;
} conf_outptr_ext;
```

Outptr Structure:

```
typedef struct {
   int32_t     errno;              /* error number on failure */
   int16_t     onscreen_x;             /* screen width in pixels */
   int16_t     onscreen_y;             /* screen height in pixels */
   int16_t     offscreen_x;            /* offscreen width in pixels */
   int16_t     offscreen_y;            /* offscreen height in pixels */
   int16_t     total_x;            /* frame buffer width in pixels */
   int16_t     total_y;            /* frame buffer height in pixels */
   int32_t     bits_per_pixel;         /* bits/pixel device has configured */
   int32_t     bits_used;          /* bits which can be accessed */
   int32_t     planes;             /* number of fb planes in system */
   uint8_t     dev_name[DEV_NAME_LENGTH];  /* null terminated product
                                                          name */
   uint32_t    attributes;         /* flags denoting attributes */
   conf_outptr_ext *ext_ptr;           /* pointer to future data */
} conf_outptr;
```

Also included in the ROM is a routine to return the current configuration of the device. This routine must return all the information required by GCDESCRIBE but may also return additional data if needed.

This routine must be passed a pointer to the following flags, via the conf_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the conf_inptr pointer:

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the conf_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- The value for onscreen_x (see GCDESCRIBE).

- The value for onscreen_y (see GCDESCRIBE).

- The value for offscreen_x (see GCDESCRIBE).

- The value for offscreen_y (see GCDESCRIBE).

- The value for total_x (see GCDESCRIBE).

- The value for total_y (see GCDESCRIBE).

- The actual bits/pixel (bits_per_pixel) which the device has been configured to.

- The number of bits (bits_used) in the bits/pixel field which are valid. See GCDESCRIBE man page for more details.

- The total number of frame buffer planes (planes) in this system.

- A null terminated string (dev_name[DEV_NAME_LENGTH]) which contains the official HP name of the product. For example, HP98705.

- A 32 bit value containing various device attributes required by GCDESCRIBE. The graphics.h and framebuf.h include files contains an up to date list of these attributes. The following revisions of these files must be used with the indicated STI ROM spec revisions.

- A pointer (ext_ptr) structure must point to the extended outptr structure and must be initialized by the caller.

This routine must return the following data in the extended output structure:

- A device dependent array of 3 integers crt_config[3]. The value of this array is determined by the various pieces of the graphics software pipeline.

- A device dependent array of 3 integers crt_hdw[3]. The value of this array is determined by the various pieces of the graphics software pipeline.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

Two values are not included, x_pitch and plane_size. These values can be calculated directly from other data. x_pitch is calculated by taking the value for total_x and muliplying it by the bits/pixel value and then dividing by eight. Both these values are returned by this routine.

Each of the crt_config[3] and crt_hdw[3] arrays are defined by the graphics software. This is a mechanism that the X11 and OGL drivers use to get device dependent information. Refer to the various graphics hardware/software ERS for a definition of each value for a specific graphics card.

5.9 set cm entry

Definition:

```
int32_t set_cm_entry(&setcm_flags,&setcm_inptr,&setcm_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                : 1;  /* should routine idle wait or not */
   uint32_t    pad                 : 31; /* pad to word boundary */
   int32_t     *future_ptr;        /* pointer to future data */
} setcm_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t     entry;              /* entry number */
   uint32_t    value;              /* entry value */
   int32_t     *future_ptr;        /* pointer to future data */
} setcm_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;          /* error number on failure */
   int32_t    *future_ptr;    * pointer to future data */
} setcm_outptr;
```

This routine is provided to allow the caller to specify the colormap used by the non-text planes, if provided by the graphics hardware.

This routine must be passed a pointer to the following flags, via the setcm_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the setcm_inptr pointer:

- The colormap entry that is to be used.

- The colormap value to be used for the corresponding entry.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the setcm_outpr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

SET_CM_ENTRY returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.10 dma ctrl

Definition:

int32_t dma_ctrl(&dma_flags,&dma_inptr,&dma_outptr,&glob_cfg)

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                : 1;   /* should routine idle wait or not */
   uint32_t    priv                : 1;   /* priv or non-priv */
   uint32_t    disable             : 1;
   uint32_t    buff                : 1;   /* buffered or un-buffered */
   uint32_t    marker              : 1;   /* write a marker */
   uint32_t    abort               : 1;   /* abort dma xfer */
   uint32_t    pad                 : 26;  /* pad to word boundary */
   int32_t     *future_ptr;           /* pointer to future data */
} dma_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t     phys_addr_upper;
   int32_t     phys_addr_lower;
   int32_t     size;
   int32_t     marker_data;
   int32_t     marker_offset;
   int32_t     *future_ptr;           /* pointer to future data */
} dma_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t     errno;          /* error number on failure */
   int32_t     *future_ptr;           /* pointer to future data */
} dma_outptr;
```

This routine allows control of dma operations.

This routine must be passed a pointer to the following flags, via the dma_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate a privileged dma operation (PRIV=1) or non-privileged operation (PRIV=0).

- A flag to indicate whether to disable (DISABLE=1) or enable (DISABLE=0) the dma operation.

- A flag to indicate the dma request is for a buffered (BUFF=1) or non-buffered (BUFF=0) dma operation.

- A flag to indicate to abort (ABORT=1) the current dma operation in progress.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the dma_intpr pointer:

- A word that contains the upper 32 bits of the physical dma address (PHYS_ADDR_UPPER).

- A word that contains the lower 32 bits of the physical dma address (PHYS_ADDR_LOWER).

- A word that identifies the size of the DMA operation in bytes (SIZE).

- A word that identifies the kernel marker data (MARKER_DATA).

- A word that identifies the kernel marker offset (MARKER_OFFSET).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the dma_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

DMA_CTRL returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.11 flow ctrl

Definition:

int32_t flow_ctrl(&flow_flags,&flow_inptr,&flow_outptr,&glob_cfg)

Flagptr Structure:

```
typedef struct {
   uint32_t   wait                  : 1;   /* should routine idle wait or not */
   uint32_t   chk_high_water        : 1;
   uint32_t   write_hw_count        : 1;
   uint32_t   write_lw_count        : 1;
   uint32_t   no_chg_cse      : 1;
   uint32_t   cs_enable             : 1;
   uint32_t   cs_write_fine   : 1;
   uint32_t   cs_write_coarse       : 1;
   uint32_t   cs_write_fifo         : 1;
   uint32_t   pad                   : 23;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t    retry_count;
   int32_t    buffer_zone;
   int32_t    high_water_count;
   int32_t    low_water_count;
   int32_t    cs_fine_val;
   int32_t    cs_coarse_val;
   int32_t    cs_fifo_count;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;          /* error number on failure */
   int32_t    retry_result;
   int32_t    fifo_size;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_outptr;
```

This routine will provide for flow control of the graphics hardware.

The host must not write to a graphics device when the device cannot accept any more data. Flow control is the term applied to the process of preventing too many writes to the graphics hardware.

For those systems that need the highest performance, graphics devices should offer interrupt paced flow control. The hardware monitors the space left in the input buffers and issues an interrupt to the host when the space becomes critically small. This can be visualized by viewing the input buffers as a water tank into which the host is pouring commands (writes) and from which the graphics pipeline is emptying commands (reads). As the host begins to write faster than the graphics pipeline reads, the water level rises. When the water level gets critically close to the top of the tank, the hardware interrupts to turn off the host spigot.

There is an additional level of flow control that STI supports. When input buffer space fills up past a "dribble water mark", the host interface may begin to slow down I/O bus cycles to reduce the rate at which the host can write data to the device. This slow down may be accomplished by stretching the current cycle with wait states or refusing it all together by issuing a "retry" or "disconnect".

Should the input buffer space fill up past a high water mark, the hardware will issue an interrupt. The interrupt is handled by the kernel and used to prevent further data from being written by the host process to the device. Obviously, the interrupt must occur while sufficient space is left in the input buffers to hold all the write data which may be in flight in the various buffers throughout the host CPU and the I/O adapter systems. The amount of data that might be written before the interrupt can reach the CPU must also be accounted for.

The current implementation assumes that flow control is available for pacing writes to the buffered path only.

This routine must be passed a pointer to the following flags, via the flow_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).
- A flag to indicate to check the high-water mark (CHK_HIGH_WATER=1).
- A flag to indicate to write the high water mark (WRITE_HW_COUNT=1).
- A flag to indicate to write the low water mark (WRITE_LW_COUNT=1).
- A flag to indicate to not change the values of the cycle stretch enable (NO_CHG_CSE=1).
- A flag to indicate to enable the cycle stretch (CS_ENABLE=1).
- A flag to indicate to write the fine adjust to the cycle stretch (CS_WRITE_FINE=1).

- A flag to indicate to write the coarse adjustment to the cycle stretch (CS_WRITE_COARSE=1).

- A flag to indicate to write the cycle stretch fifo (CS_WRITE_FIFO=1).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the flow_inptr pointer:

- A word that indicates the retry_count (RETRY_COUNT). This is the number of times a check of the high water mark will be attempted.

- A word that indicates the buffer zone (BUFFER_ZONE). This is an indication of the number of words of data that might be inflight in the graphics pipeline and not posted to the fifo.

- A word that indicates the high water count to write (HIGH_WATER_COUNT).

- A word that indicates the low water count to write (LOW_WATER_COUNT).

- A word that indicates the cycle stretch fine value to write (CS_FINE_VAL).

- A word that indicates the cycle stretch coarse value to write (CS_COARSE_VAL).

- A word that indicates the cycle stretch fifo depth (CS_FIFO_COUNT).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the flow_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A value that indicates the retry result failed and the fifo has not emptied out (RETRY_RESULT=1) or the fifo has emptied out (RETRY_RESULT=0).

- A value that indicates the current size of the fifo (FIFO_SIZE).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

FLOW_CTRL returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.12 user timing

Definition:

```
int32_t user_timing(&timing_flags,&timing_inptr,&timing_outptr,&glob_cfg)
```

Flagptr Structure:

```
typedef struct {
   uint32_t   wait                  : 1;   /* should routine idle wait or not */
   uint32_t   kbuf_size             : 1;   /* this call has kbuf_size */
   uint32_t   pad                   : 30;
   int32_t    *future_ptr;          /* pointer to future data */
} timing_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t    *data;
   int32_t    *kbuf;
   int32_t    *future_ptr;          /* pointer to future data */
} timing_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t    errno;          /* error number on failure */
   int32_t    kbuf_size;            /* size we are requesting from kernel in
                                       bytes */
   int32_t    *future_ptr;          /* pointer to future data */
} timing_outptr;
```

This routine will allow the user to pass in a new set of timing data.

This routine must be passed a pointer to the following flags, via the timing_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate that the call contains the kbuf size (KBUF_SIZE=1) or does not contain the kbuf size (KBUF_SIZE=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the timing_inptr pointer:

- A word aligned pointer to the data buffer (data).

- A word aligned pointer to the kernel buffer (kbuf).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the timing_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A word that indicates the size of the kernel buffer required (KBUF_SIZE).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

USER_TIMING returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.13 process mgr

Definition:

```
int32_t
        process_mgr(&process_flags,&process_inptr,&process_outptr,
                    &glob_cfg)
```

Opcode Definition:

```
enum OPCODE {
    cleanup             = 0,
    buff_access_ctrl            = 1
  };
```

Flagptr Structure:

```
typedef struct {
   uint32_t    wait                : 1;    /* should routine idle wait or not */
   enum OPCODE opcode              : 4;
   uint32_t    critical            : 1;
   uint32_t    buff                : 1;
   uint32_t    ignore_buff         : 1;
   uint32_t    pad                 : 24;
   int32_t     *future_ptr;        /* pointer to future data */
} process_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t     reserved[4];
   int32_t     *future_ptr;        /* pointer to future data */
} process_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t     errno;          /* error number on failure */
   int32_t     *future_ptr;        /* pointer to future data */
} process_outptr;
```

This routine allows for kernel control of the device and is associated with cleanup of operations and processes that have become hopelessly confused. This is typically used with hardware devices that support indirect reads.

Indirect Reads are really a process and not a single operation. Normal reads are performed by a single load instruction. For indirect reads, the target registers to be read are not accessed by a load instruction. Instead, a flag is set to tell the hardware to convert writes to reads. Once this flag is set, the target registers are written with bogus data. The hardware converts the writes into reads and retrieves the contents of the target registers, storing them in a read FIFO. Unbuffered indirect read data are stored in the unbuffered read FIFO while buffered indirect read data are stored in the buffered read FIFO. The host may acquire the data by reading special registers that return data from one of these FIFOs.

On most hardware implementations, the host may also enable the DMA controller to return indirect read results to prescribed locations in main system memory. Either path may use DMA to return data.

Whenever something goes wrong with a process which is in control of one of the graphics device pipelines it may be necessary to prematurely rid the given pipeline of the indirect reads which have been queued by the process.

It is presumed that a purge will cause:

1. The host interface to launch a purge marker down the given pipeline and will signal the DMA block to start a purge operation.

2. The DMA controller will discard any indirect read data which is returned from the given pipeline until it receives the purge marker. Data that is targeted for DMA return to host memory is not affected.

3. The DMA controller will assert its busy signal until it receives the purge marker.

4. DMA operations will not be affected.

Normally, this routine is called with either the opcode set to buff_access_ctrl or opcode set to cleanup. In the buff_access_ctrl mode, the ignore_buff flag indicates whether to set the hardware to a state where it ignores all writes and returns random data on reads, or whether it responds normally. It is presumed that the hardware is in a confused state and sending additional writes or reads would not yield any predictable behavior. The hardware must still respond to bus activity and the graphics hardware will continue to handshake transactions while in this mode.

In the opcode = cleanup mode, the kernel can request the device the clear out the “write to read” path, for either the buffered path or the unbuffered path. Additionally, the routine might reset certain parts of the device and flush any buffered and unbuffered operations that remain pending.

Typical usage would be for the caller to first call with the buf_access_ctrl opcode and the flag to ignore_buff. This would put the hardware into a handshake only mode for buffered operations, effectively locking the device out from further transactions that would be meaningless. The caller would then call the routine with the opcode = cleanup and the appropriate flags set for either the buffered or unbuffered path. This would clear up the current path. The caller would then call one last time with buf_access_ctrl mode to re-enable the hardware to respond by clearing the ignore_buff flag.

This routine must be passed a pointer to the following flags, via the process_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A 4 bit encoded value that indicates the opcode to execute (OPCODE=value). Currently there are only 2 defined opcodes, cleanup (value=0) and buff_access_ctrl (value=1).

- A flag to indicate this is considered a critical section part of the kernel (CRITICAL=1) or not (CRITICAL=0).

- A flat to indicate this is control of the buffered path (BUFF=1) or the unbuffered path (BUFF=0). This is normally used in conjunction with the OPCODE=cleanup.

- A flag to indicate to the hardware to ignore buffered transactions (IGNORE_BUFF=1) or to respond in a normal manner (IGNORE_BUFF=0). Normally, the hardware will have a mechanism to handshake transactions on the bus and will bit bucket writes and return undefined data on reads. This access mode is provided to help make it easier to clean up a process that has become hopelessly lost.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the process_inptr pointer:

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must return the following data in the output structure, via the process_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

PROCESS_MGR returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

5.14 sti util

Definition:

int32_t sti_util(&util_flags,&util_inptr,&util_outptr,&glob_cfg)

Flagptr Structure:

```
typedef struct {
   uint32_t    root_user                : 1;    /* ioctl(GC_STI_UTIL) called as
                                                        root */
   uint32_t    pad                      : 31;
   int32_t     *future_ptr;         /* pointer to future data */
} util_flags;
```

Inptr Structure:

```
typedef struct {
   int32_t     in_size;            /* size of the incoming data */
   int32_t     out_size;           /* size of the outgoing data */
   uint8_t     *buffer;            /* incoming/outgoing data */
} util_inptr;
```

Outptr Structure:

```
typedef struct {
   int32_t     errno;              /* error number on failure */
   int32_t     *future_ptr;        /* pointer to future data */
} util_outptr;
```

This routine allows for generic usage of the device by user processes. This routine may exist for certain classes of hardware, and not for others. See the example code in Chapter 6 for one method of using this routine.

From a user process, the GC_STI_UTIL ioctl() is used to pass data and receive return data. Typically, this routine has been used to allow user mode processes, such as a graphics driver to work around hardware problems and it has also been used to allow a user mode process to request a re-flash of the STI ROM, when that ROM is a re-programmable device.

This routine must be passed a pointer to the following flags, via the util_flags pointer:

- A flag to indicate whether the ROM routine should wait in an idle loop for the graphics device to finish its operation (WAIT=1) or return to the calling program (WAIT=0).

- A flag to indicate that the caller has superuser privileges (ROOT_USER=1) or not (ROOT_USER=0).

- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

This routine must be passed a pointer to a structure containing the following, via the util_inptr pointer:

- A word that indicates the incoming data size (IN_SIZE).
- A word that indicates the returning data size (OUT_SIZE).
- A pointer the incoming and outgoing buffer.

This routine must return the following data in the output structure, via the util_outptr pointer:

- An error number that will contain an error value if a failure has occurred within this routine.
- A pointer (future_ptr) that can be used for adding an additional structure in the future if needed.

STI_UTIL returns 0 if successful, -1 if failure, 1 if it exited rather than waited in an idle loop.

6 Usage

6.1 Routine usage considerations

Because STI is intended to be an interface for use by a variety of systems, operating systems and graphics devices, it is necessary to insure that the number of assumptions which developers and users will be making is minimized. This section is therefore provided to document various constraints on the usage of STI routines that must be adhered to.

The INIT_GRAPH routines must always be the first routine executed when starting to use a device with STI support. The kernel has to be especially careful to insure that INIT_GRAPH has been executed prior to the time when the kernel's interrupt service routine (ISR) can first be invoked. If this is not done, the ISR may try to execute the EXCEP_HDLR routine, which will fail because the glob_conf structure has not been set up. The kernel must also manage the glob_conf structure for each STI device so that any parts of the kernel which want to use an STI routine can pass the appropriate glob_conf structure for that device to the STI routine. When INIT_GRAPH is first called, it must have the hard reset flag set so that a hard reset is done. Prior to calling INIT_GRAPH, the memory region array (in glob_conf) must contain the starting addresses (virtual or physical, as appropriate) for each region, and reent_lvl must be set to zero by the calling routine.

INIT_GRAPH must ignore the state of the wait flag if the hard reset flag is set.

INIT_GRAPH should do the clear (if requested) as early as possible in the routine to insure that the user does not get anymore "trash" on the screen than is necessary.

SELF_TEST is the only STI routine that might leave the graphics device in an indeterminate or "trashed" state. Therefore, INIT_GRAPH must be called sometime prior to executing SELF_TEST and then must be called again immediately after SELF_TEST completes. The call immediately after SELF_TEST must have the hard reset flag set. This must be done promptly for all STI devices even if they will not be the ITE since all STI devices may be accessed by such STI routines as EXCEP_HDLR and INQ_CONF. In particular, a call to the EXCEP_HDLR routine may come at any time after interrupts have been enabled and INIT_GRAPH with hard reset must have been performed before then.

The STATE_MGMT routine must always be called by the kernel to do a state save before the any other STI routine is called, with the following exceptions.

- Saving state is not necessary and should not be done, if INIT_GRAPH is being called with the hard reset flag set.

- Saving state is not necessary if EXCEP_HDLR is the STI routine being accessed.

If a state save has been previously done by the STATE_MGMT routine, then a complementary state restore must be done at some later time (unless a hard reset is done with INIT_GRAPH). Pairs of state saves/restores may be nested as long as each state save uses a different location in which to save the state.

The STATE_MGMT routine must not save or restore the displayable state of the text and non-text planes. This is because the ITE may be asked by the user to change this state (e.g., turn on the text planes), and such changes need to remain in place after the ITE has exited. If such state is restored after the ITE has run, then the change which the ITE just made could be undone. Unless calling routines have a reason to request either a change in the displayability of the text or non-text planes, or a change in the state of the bus error timer or interrupt, the routines should use the flags provided by INIT_GRAPH for requesting no change on these items.

The FONT_UNP/MV routine must be written such that the actual ASCII value of the character to be displayed can be passed in. FONT_UNP/MV must then account for the situation in which the font table being used does not start with character zero. To do this, the routine should take the value for the ASCII character that it has been passed and subtract from it the value for the ASCII character that is the first character in the font. The routine will then have a value which can used as an index into the font table.

If the EXCEP_HDLR routine has saved the state of the interrupt masks and disabled the interrupts, it must be called to restore the state of the masks before it is again asked to save them.

STI code developer must include a check to see if a bus error has occurred while they are waiting in a loop for an event. This is to avoid the potential for the STI routine becoming hung because of a bus error. STI code developers should also make sure that common hardware configuration mistakes that the customer might make (such as not powering up part of the graphics device), will not cause the
INIT_GRAPH routine to hang but return an error code instead.

All code developers, calling routines and STI routines, should make sure that all unused locations are set to, and remain zero. This is necessary to preserve to ability to use these locations in the future should it become necessary. This includes all pointers, unused flags (padded) in flag words and unused ROM locations. Calling routines may want to structure their routine such that data locations are initialized to zero when they are first allocated.

Calling routines may need to flush the cache after they have loaded the STI routine from ROM into RAM. Check with your local experts.

6.2 Boot ROM usage

Developers interested in using STI in system firmware must consult their appropriate firmware architecture ERS.

The only unique support that STI has for system firmware is that is supports a few limitations of the current PDC/IODC restrictions. The basic restriction is that no single STI routine can be larger than 10k bytes. The console IODC is limited to 16k bytes and is split between the STI interface and the keyboard interface. The STI code is limited to 10k bytes, leaving 6k bytes for the keyboard IODC.

STI code writers should take special care to limit the combined size of the font_unpmv and the block_move routines to a total of 10k bytes. This allows the system firmware to have both loaded simultaneously. If this is not the case, scrolling at boot time can be extremely slow, as the system firmware must alternate between loading and running these two routines.

The following is a list of routines accessed by the Boot ROM:

1. init_graph
2. self_test
3. inq_conf
4. block_move
5. font_unpmv
6. set_cm_entry

In order to take advantage of STI code from the boot ROM you must use the ODE developer kit that is available in a separate document. ODE provides an environment that allows complete control over all hardware resources.

6.3 Kernel usage

The following is a list of routines accessed by the HP-UX Kernel:

1. init_graph
2. inq_conf
3. block_move
4. font_unpmv
5. set_cm_entry
6. state_mgmt
7. exception_hdlr
8. dma_ctrl
9. flow_ctrl
10. user_timing
11. process_mgr
12. sti_util

As you can see, the Kernel calls all routines except for self_test.

6.3.1 iomap

When no STI ROM exists you can use the iomap capabilities of HP-UX to map the hardware into user address space. This is only when there is no STI ROM support. If there is STI ROM support you must use gcmap.

In order to use iomap, you must first add the iomap driver to your /stand/system file and regenerate the kernel. You must then make a device file, see the iomap(7) man page for information on how to do this. Once you have the iomap device file you can address the device. Below is a trivial example program to map in a device.

```
#include <stdio.h>
#include <sys/framebuf.h>
#include <fcntl.h>
#include <malloc.h>
#include <sys/iomap.h>
#include <errno.h>

char *null = NULL;
extern int errno;
char *errorstring();
extern char *sys_errlist[]; /* list of error msgs          */
extern int sys_nerr;

main(argc, argv)
int       argc;
char      *argv[];
{
char      *outdev, device_file[256];
int       filedes,i;
char *attach_addr;


          attach_addr = NULL;

   outdev = (char *)getenv("STI_DEV");

   if (outdev == NULL)
      outdev = "/dev/crt";

   strcpy(device_file, outdev);
```

```
   if ((filedes = open(device_file, O_RDWR, 0)) == -1) {
            perror("open");
                  exit(-1);
         }

         if (ioctl(filedes,IOMAPMAP,&attach_addr) < 0) {
      fprintf(stderr,"cannot iomap device %s at address 0x%08x: %s\n",
         outdev,attach_addr, errorstring(errno,sys_nerr,sys_errlist));
      exit(1);
   }

         /* attach_addr has the address you use to talk to the device */

}
char *
errorstring(err,nerr,stgs)
    int err;          /* errno value         */
    int nerr;         /* size of error string table   */
    char *stgs[];     /* error string table       */
{
   static char buf[40];    /* buffer to make new error stg */

   /* if this error isn't in the string table, make one ourselves */
   if (err < 0 || err > nerr) {
   sprintf(buf,"ERROR(local): %d\n",err);
   return(buf);
   }
   else
   return(stgs[err]);
}
```

6.3.2 gcmap

When there is an STI ROM then you use gcmap to map the device to user space.
Included below is sample code that will call the GCRESET ioctl() to request the kernel to reset the device.

```
/*
   includes
*/

#include <stdio.h>
#include <sys/framebuf.h>
```

```
#include <fcntl.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>

/*
   externs
*/

extern char *sys_errlist[]; /* list of error msgs          */
extern int  sys_nerr;       /* size of sys_errlist         */
extern int  errno;          /* error syndrome number       */
extern int  getopt (        /* library function getopt(3C) */
          int argc,
          char *const argv[],
          const char *optstring
          );
extern int  optind;
extern char *optarg;

/*
   forward declarations
*/

void        Usage ();

/*
   defines
*/

/*
   structs
*/

/*
   globals
*/

crt_frame_buffer_t     data_about_crt;
char                        *null = NULL;
char                   *progname;         /* name of this program */
```

```
int                                     filedes;


int
main(argc, argv)
int       argc;
char      *argv[];
{
char                    device_file[256];
char              *hpa;
int                     i;
int                             speed,val,temp;
unsigned int      *addr;

   progname = *argv;
#if 0
   while ((i = getopt(argc,argv,"o:?")) != EOF)
   switch (i)
   {
      case 'o':
                        speed = strtoul(optarg,NULL,0);
         break;
      case '?':
      default:
         Usage();
   }

   /*
      did we find all the options ?
   */

   if (optind > argc)
   {
      for (i=optind;i<argc;i++)
         fprintf(stderr,"%s: illegal cmd line argument -- %s\n",
            progname,argv[i]);

      Usage();
   }


         /*
                  remaining option is dev file
         */

   strcpy(device_file, argv[optind]);
```

```
#endif
   strcpy(device_file, argv[1]);
        printf("dev file = %s\n",argv[1]);

        /*
                figure out which clock entry to set based on input
        */

   if ((filedes = open(device_file, O_RDWR, 0)) == -1)
        {
           perror("open");
                exit(-1);
        }

   if (ioctl(filedes, GCMAP, &null) == -1)
        {
                perror("GCMAP");
                exit(-1);
        }

   if (ioctl(filedes, GCRESET, &null) == -1)
        {
                perror("GCRESET");
                exit(-1);
        }
}

void
Usage ()
{
   register char **s;
   static char *stgs[] =
   {
      "where options include:\n\n",
      "    -?         this menu\n",
      NULL
   };

   static char first_line[256];


   sprintf(first_line,"\nUsage: %s [-options] device_file\n",progname);
   fputs(first_line,stderr);

   for(s=stgs; *s != NULL; s++)
      fputs(*s,stderr);
```

```
    exit(-1);
}
```

6.4 Millicode

Millicode is supported. You will not want to use the system supplied millicode library, as it is too large. There is a special millicode library included with the STI toolkit that has all the interesting millicode you will need.

6.5 Floating Point Usage

The floating point unit is not initialized at boot time thus no usage of floating point is allowed in any of the routines that are accessed via the Boot ROM. See the Boot ROM usage section for details on which routines are used by the Boot ROM.

6.6 User Space usage

6.6.1 Sample Util usage

An ioctl() that allows direct access to the STI ROM has been added. The ioctl is called GCSTI_UTIL. It is a '_IO' type ioctl. The definition is

```
#define GCSTI_UTIL_IO('G',53)
```

The parameter is a structure defined as

```
typedef struct
{
        int in_size;
        int out_size;
        char buffer[24];
}
```

Where :

in_size: The size, in bytes, of the entire structure that is being passed into the ioctl. This size includes the bytes for both in_size and out_size (8 bytes). When this ioctl is called, the kernel will copy the first in_size bytes from gc_util_t and put them into the first in_size bytes of the STI util_inptr structure.

out_size: The size, in bytes, that is the maximum buffer size that the STI ROM will be returning back to the user. This size includes the bytes for both in_size and out_size. When this ioctl is called, the kernel will compare in_size and

out_size. The larger of the two values wins and this is the size of the internal buffer that the kernel will allocate.

buffer: The data that is passed into the STI ROM, and the data that is passed back from the STI ROM. When passing data into the STI ROM, the size of the buffer is (in_size – 8). When passing data back from the STI ROM, the size of the buffer is (out_size – 8). The size of the buffer is the maximum of in_size and out_size.

There will always be a minimum structure size. This is the structure as defined above. The minimum size is 32 bytes, which is the same as a cache line. This is equivalent to in_size, out_size and 24 bytes of data passed into the buffer.

The kernel needs to do a data transfer to get the value of in_size. Once it has in_size it can then transfer the rest of the data over. This is two data transfers. By having a minimum size the kernel will always pull across 32 bytes of valid data. Now it can check in_size. If in_size is 32 or less the kernel is done. If in_size is 32 or less, the kernel is done. It does not need to transfer over any more data.

A basic description of what happens when this ioctl is called :

```
USER_CODE
        Gc_util_t.in_size = size of input buffer;
        Gc_util_t.out_size = maximum size of output buffer;
        Gc_util_t.buffer = input buffer;
        Ioctl(fd,GCSTI_UTIL, &gc_util_t);

KERNEL CODE
        Copy over the first 32 bytes from gc_util_t into util_inptr
        Buf_size = maximum of (in_size and out_size)
        If (buf_size >32)
        Then
                Reallocate util_inptr to be buf_size bytes
                Copy over the first in_size bytes from gc_util_t into util_inptr
        Endif
        Call STI routine sti_util, passing in inptr

STI CODE
        If there is an error
        Then
                Set util_outptr.errno
        Else
                Update data in util_inptr.buffer
                Update size of data in util_inptr.out_size
        Endif
        Return to kernel
```

KERNEL CODE

```
If  (util_outptr.errno != 0)
Then
        Set errno to util_outptr.errno
        Return –1
Else
        If (util_inptr.out_size is larger than original out_size)
        Then
                Set errno
                Return –1
        Else
                Copy the first util_inptr.out_size bytes from
                        util_inptr.buffer into gc_util_t.buffer
                Return 0
        Endif
Endif
```

RETURN VALUES

A value of 0 means that the STI routine completed successfully. If there are any problems then return –1 with errno set to one of the following :

EACCESS: The caller did not own the graphics lock when calling this ioctl

EPERM: We are trying to run code from the buffer and wer are not ROOT

ENOMEM: Either in_size or out_size is too large and we are not able to allocate the memory for it.

EINVLA: The output buffer size returned by the STI ROM is larger than specified by gc_util_t.out_size.

Examples of both the user code and STI implementation follow.

6.6.1.1 User Code

```
/*
   includes
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
#include <sys/mman.h>
#include <errno.h>
#include <sys/framebuf.h>
#include <fcntl.h>
#include <string.h>
#include <ctype.h>
#include <strings.h>


/*
   externs
*/

extern char *sys_errlist[]; /* list of error msgs          */
extern int  sys_nerr;       /* size of sys_errlist         */
extern int  errno;          /* error syndrome number       */
extern int  getopt (        /* library function getopt(3C)  */
          int argc,
          char *const argv[],
          const char *optstring
          );
extern int  optind;
extern char *optarg;

/*
   forward declarations
*/

void        Usage ();

/*
   globals
*/

crt_frame_buffer_t     data_about_crt;
char                        *null = NULL;
char                   *progname;         /* name of this program */


int
main(argc, argv)
int       argc;
char     *argv[];
{
char                   device_file[256];
int                    i;
```

```
unsigned char  *addr;
int                              error;
int                              filedes;
gc_util_t                test_data;
unsigned int    *opcode;
unsigned int    *param1;


        printf("\n");

   progname = *argv;

        while ((i = getopt(argc,argv,"?")) != EOF)
   switch (i)
   {
      case '?':
      default:
         Usage();
   }

        if (argc < 2)
                Usage();

        if (optind > 1)
   {
      for (i=optind;i<argc;i++)
         fprintf(stderr,"%s: illegal cmd line argument -- %s\n",
            progname,argv[i]);

      Usage();
   }

   strcpy(device_file, argv[optind++]);
        printf("dev file = %s\n",device_file);

        /*
                open up device
        */
   if ((filedes = open(device_file, O_RDWR, 0)) == -1)
        {
           perror("open");
                exit(-1);
        }

   if (ioctl(filedes, GCMAP, &null) == -1)
        {
```

```
                perror("GCMAP");
                exit(-1);
        }

        if (ioctl(filedes, GCDESCRIBE, &data_about_crt) == -1)
{
   perror("GCDESCRIBE");
   exit(-1);
}


        /*
                check for My Hardware
        */
if (data_about_crt.crt_id != <the low 32 bits of your graphics id>)
        {
    printf("The device %s is not a My Hardware\n",device_file);
                exit(1);
        }
        else
                printf("Found My Hardware in %s\n",device_file);

        /*
                make the opcode data
        */

        test_data.in_size = 16;  /* insize,outsize,opcode,param1 */
        test_data.out_size = 16;
        opcode = (unsigned int *)&test_data.buffer[0];
        param1 = (unsigned int *)&test_data.buffer[4];
        *opcode = 1;   /* opcode == 1 is current mon */

        ioctl(filedes, GCLOCK, 0);

if (error = ioctl(filedes, GCSTI_UTIL, &test_data) != 0)
        {
                perror("GCSTI_UTIL");
        printf("errno = %d\n",error);
                exit(-1);
        }
        else
        {
                printf("current mon = %d\n",*param1);
        }

        ioctl(filedes, GCUNLOCK, 0);
```

```
}

void
Usage ()
{
   register char **s;
   static char *stgs[] =
   {
      "where options include:\n\n",
      "    -?        this menu\n",
      NULL
   };

   static char first_line[256];


   sprintf(first_line,"Usage: %s [-options] device_file\n\n",progname);
   fputs(first_line,stderr);

   for(s=stgs; *s != NULL; s++)
      fputs(*s,stderr);

   exit(-1);
}
```

6.6.1.2 STI Code

```
#include "std_main.h"

/*
   special util structure

         the minimum passed to the kernel is (32 bytes) :

                  typedef struct
                  {
                           int in_size;
                           int out_size;
                           char buffer[24];
                  } gc_util_t;

         and the mininum passed from kernel to STI is our util_struct shown below
```

```
*/

typedef struct {
   int32_t    opcode;
   int32_t    param1;
   int32_t    param2;
   int32_t    param3;
   int32_t    param4;
   int32_t    param5;
} util_struct;

#if 0    /* { */

        this routine allows us to do just about anything

        the opcodes I have defined are
        ------------------------------
                0                       - NOP
                1                       - return current monitor

        opcode 1 definition
        -------------------

                input
                -----

                -------------
                | opcode  |       = 1
                -------------

                output
                ------

                outsize = 16    includes insize,outsize and 2 ints in buffer

                -------------
                | opcode  |     = 1
                -------------
                | param1  |     = current monitor
                -------------

#endif /* } */


/*****************************************************************
 *
```

```
 * NAME
 *  sti_util
 *
 * PURPOSE
 *  Perform various utility functions
 *
 * RETURN VALUE
 *   0    pass
 *  -1    fail
 *   1    not ready
 *
 ***************************************************************/

int32_t
sti_util (
        util_flags              *flagptr,       /* input flags  */
        util_inptr              *inptr,         /* input data   */
        util_outptr             *outptr,        /* output data  */
        glob_cfg                *configptr      /* device config*/
        )
{

        /*
                required declarations for use by common macros
        */

   DECLARE_BASE;
   DECLARE_CALL_ERROR;
   DECLARE_RET_ERROR;

   /*
                declare flagptr variables to save codespace
   */

        int32_t         root_user;

   /*
                declare inptr variables to save codespace
   */

        int32_t         in_size;
        int32_t         out_size;
        uint8_t         *buffer;

        /*
      declare outptr variables to save codespace
```

```
   */

   /*
                declare configptr variables to save codespace
   */

   /*
                hardware registers
   */

   /*
                declare register temps
   */

        /*
      declare local variables
   */

        util_struct     *util_data;
        int32_t         opcode;

/*****************************************************************
 *
 * initialization
 *
 *****************************************************************/

#ifdef TEST /* { */

   /*
                make sure that flagptr,inptr,outptr,configptr
                are valid
   */

   VALID_PROC_CALL;

   /*
                make sure that we got valid region pointer to get
                to hardware
   */

   VALID_CALLER;

#endif /* } */

   /*
```

```
                setup hpa, for all macros, etc. that use it as base
                with which to add reg offset
    */

    SETUP_BASE;

    /*
                just in case there is anything unique to be done on
                entry to any STI routine
    */

    ENTRY_TO_STI;

/*****************************************************************
 *
 * boot rom debugging (output to RS232)
 *
 *****************************************************************/

    /*
       this routine is not used by boot rom thus no
       boot debug capability
    */

/*****************************************************************
 *
 * variable setup before code reentry
 *
 *****************************************************************/

/*****************************************************************
 *
 * check for code reentry
 *
 *****************************************************************/

        STD_REENTRY_CHECK;

        /*
                anything that needs to be skipped on code reentry
                should be put between the STD_REENTRY_CHECK and
                STD_REENTRY_LABEL macros
        */

        STD_REENTRY_LABEL:
```

```
/*************************************************************
 *
 * wait for hardware ready
 *
 *************************************************************/

/*************************************************************
 *
 * non-dependent internal variables
 *
 *************************************************************/

/*************************************************************
 *
 * internal variables
 *
 *************************************************************/

  /*
        initialize local variables to save code space
  */

      root_user     = (flagptr->root_user)?1:0;
      buffer        = (uint8_t *)&inptr->buffer;
      util_data     = (util_struct *)&inptr->buffer;
      in_size       = (inptr->in_size);
      opcode        = util_data->opcode;

/*************************************************************
 *
 * error check
 *
 *************************************************************/

/*************************************************************
 *
 * main body
 *
 *************************************************************/

      /*
        switch/case construct may be preferrable in terms
        of code readability, but no guarantee I will compile PIC
      */

      if (opcode == 0)                  /* NOP */
```

```
    {

            /*
                    we return no data
            */

            out_size = 0;

    }
    else if (opcode == 1)   /* current monitor */
    {
            uint8_t current_mon;

            current_mon = (uint8_t)READ_CURR_MON(csp);

/*
    we return current monitor
*/

out_size = 16;
util_data->param1 = current_mon;

    }
    else
    {

            /*
                    Don't recognize that opcode.

                    By definition, on failure the kernel will disregard
                    out_size, but we set it to 0 anyway.

                    By definition, the kernel will pass back the errno
                    to user code;
            */

            inptr->out_size = 0;

            /*
                    now we can bail
            */

            ABNORMAL_RETURN(INVALID_UTIL_OPCODE);

    }
```

```
        /*
                always set the outsize in the buffer passed in
        */

        inptr->out_size = out_size;

/*****************************************************************
 *
 * done
 *
 *****************************************************************/

        NORMAL_RETURN;

/*****************************************************************
 *
 * std exit
 *
 *****************************************************************/

   EXIT_LABEL:

   EXIT_FROM_STI;
}
```

6.6.2 Sample DMA usage

6.6.2.1 User Code

DMA Skeleton code

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/framebuf.h>

#define BUFFER_SIZE (32 * 1024)  /* 32K */

main()
{
        int32_t         fd;                             /* file device */
        uint32_t        *dmaBuffer;                 /* local dma buffer */
        gc_dma_t        gcDma;                          /* kernel dma struct */
        uint32_t        *csAddr;                    /* pointer to control space */
        uint32_t        *fbAddr;                    /* pointer to frame buffer */
        crt_frame_buffer_t gcInfo;              /* kernel device info struct */
```

```
/* open the device */
fd = open("/dev/crt",O_RDWR)

/* map device into this process */
ioctl(fd,GCMAP,&addr)

/* get address pointers */
ioctl(fd,GCDESCRIBE,&gcInfo)
csAddr = (uint32_t *) gcInfo.crt_control_base;
fbAddr = (uint32_t *) gcInfo.crt_frame_base;

/* start critical section */
ioctl(fd,GCLOCK,0);

/* read device to make sure its ready for dma setup */

<pseudo code> check hardware here for dma ready using csAddr and fbAddr

/* end critical section */
ioctl(fd,GCUNLOCK,0);

/* setup dma */
gcDma->address = NULL;
gcDma->flags = (gc_dma_flags_t) 0;
gcDma->size = BUFFER_SIZE;
ioctl(fd,GCDMA_ALLOC,gcDma);
ioctl(fd,GCDMA_SET,gcDma);

/* get the dma buffer address */
dmaBuffer = gcDma->address;

/* fill out buffer with dma data */
*dmaBuffer++ = 1;
*dmaBuffer++ = 2;
*dmaBuffer++ = 3;
*dmaBuffer++ = 4;

/* trigger dma */
/* start critical section */
ioctl(fd,GCLOCK,0);

/* write dma trigger to hardware */

<pseudo code> write trigger to hardware using csAddr and fbAddr
```

```
        /* end critical section */
        ioctl(fd,GCUNLOCK,0);

        /* wait for DMA complete */
        ioctl(fd,GCDMA_WAIT,gcDma);

        /* close device */
        ioctl(fd,GCUNMAP,0);
}
```

6.6.2.2 STI Code

```
#include "std_main.h"

#if 0 /* { */

   ************************************************************
   STI Interrupt structure from sti.h
   ************************************************************

   typedef struct {
      uint32_t    bet     : 1;    /* bus error timer */
      uint32_t    hw      : 1;    /* high water */
      uint32_t    lw      : 1;    /* low water */
      uint32_t    tm      : 1;    /* texture map */
      uint32_t    vb      : 1;    /* vertical blank */
      uint32_t    udc     : 1;    /* un-buffered dma complete */
      uint32_t    bdc     : 1;    /* buffered dma complete */
      uint32_t    udpc    : 1;    /* un-buffered priv dma complete */
      uint32_t    bdpc    : 1;    /* buffered priv dma complete */
      uint32_t    pad     : 23;
   } int_desc;

#endif /* } */

typedef union
{
   uint32_t all;
   int_desc bits;  /* int_desc is defined in sti.h */
   struct {
      uint32_t ALL_INTS   : 9;
      uint32_t pad1       : 23;
   } joined;
}int_field;
```

```
/*****************************************************************
 *
 * NAME
 *  excep_hdlr
 *
 * PURPOSE
 *  Perform exception handling
 *
 * RETURN VALUE
 *   0  pass
 *  -1  fail
 *   1  not ready
 *
 *****************************************************************/

int32_t
excep_hdlr (
   excep_flags     *flagptr,   /* input flags  */
   excep_inptr     *inptr,     /* input data   */
   excep_outptr    *outptr,    /* output data  */
   glob_cfg        *configptr  /* device config*/
   )
{

   /*
      required declarations for use by common macros
   */

   DECLARE_HPA;
   DECLARE_CALL_ERROR;
   DECLARE_RET_ERROR;

   /*
      declare flagptr variables to save codespace
   */

   /*
      declare inptr variables to save codespace
   */

   excep_inptr_ext     *ext_inptr;
   uint32_t            *int_save_addr;
   int_field           int_emask;      /* wrapped STI int structure */
   int_field           int_clear_mask; /* wrapped STI int structure */
```

```
    /*
       declare outptr variables to save codespace
    */

    excep_outptr_ext    *ext_outptr;
    int_field           int_pend;       /* wrapped STI int structure */

    /*
       declare configptr variables to save codespace
    */

    /*
       hardware registers
    */

    status_reg          status;
    control_reg         control;

    /*
       declare register temps
    */

    /*
       declare local variables
    */

    int32_t             fault_active;
    uint32_t            global_enable;
    uint32_t            int_temp, int_enables;

/*****************************************************************
 *
 * initialization
 *
 *****************************************************************/

#ifdef TEST /* { */

    /*
       make sure that flagptr,inptr,outptr,configptr
       are valid
    */

    VALID_PROC_CALL;

    /*
```

```
        make sure that we got valid region pointer to get
        to hardware
    */

    VALID_CALLER;

#endif /* } */

    /*
        setup hpa, for all macros, etc. that use it as base
        with which to add reg offset
    */

    SETUP_HPA;

    /*
        just in case there is anything unique to be done on
        entry to any STI routine
    */

    ENTRY_TO_STI;

/*****************************************************************
 *
 * boot rom debugging (output to RS232)
 *
 *****************************************************************/

    /*
        this routine is not used by boot rom thus no
        boot debug capability
    */

#ifdef KERNEL_DEBUG_EXCEP /* { */

#   include "BOOT/b_e"

#endif /* } */

/*****************************************************************
 *
 * variable setup before code reentry
 *
 *****************************************************************/

/*****************************************************************
```

```
 *
 * check for code reentry
 *
 ***************************************************************/

   STD_REENTRY_CHECK;

   /*
      anything that needs to be skipped on code reentry
      should be put between the STD_REENTRY_CHECK and
      STD_REENTRY_LABEL macros
   */

   STD_REENTRY_LABEL:

/***************************************************************
 *
 * wait for hardware ready
 *
 ***************************************************************/

   /*

      Normally we would do HDW_CHECK_DEVICE_WAIT(1) here
      but we handle the hardware in a different manner
      in excep_hdlr than in other routines

      For excep_hdlr, we are going to access the hardware
      Unbuffered regs, so we will not do any busy waiting... EVER

   */

/***************************************************************
 *
 * non-dependent internal variables
 *
 ***************************************************************/

   global_enable = 0;

/***************************************************************
 *
 * internal variables
 *
 ***************************************************************/
```

```
    /*
       initialize local variables to save code space
    */


    /*
       pointer to interrupt enables save address
    */

    int_save_addr = (uint32_t *) inptr->save_addr;

#ifdef TEST /* { */

    ext_inptr = inptr->ext_ptr;
    ext_outptr = outptr->ext_ptr;

#endif /* } */

/*****************************************************************
 *
 * error check
 *
 *****************************************************************/

#ifndef TEST /* { */

    /*
       if both save and restore int enable flags are set
       then error
    */

    if (flagptr->save_int && flagptr->restore_int)
    {
       ABNORMAL_RETURN(INVALID_EXCEP_FLAGS, 0, FAIL);
    }

    /*
       we must always do these checks because an
       invalid pointer may mess up the kernel and
       a NULL pointer will guarantee an HPMC
    */

    /*
       if we are asked to either save or restore interrupt
       states and the address passed is NULL then error
    */
```

```
   if ((int_save_addr == NULL)
      && (flagptr->save_int || flagptr->restore_int))
   {
      ABNORMAL_RETURN(INVALID_EXCEP_ADDR, 0, FAIL);
   }

   /*
      check for extended structures
   */
   if (ext_outptr == NULL)
   {
      ABNORMAL_RETURN(NO_EXCEP_HDLR_EXT, 0, FAIL);
   }

   if (ext_inptr == NULL)
   {
      ABNORMAL_RETURN(NO_EXCEP_HDLR_EXT, 0, FAIL);
   }

   /*
      these are valid so we can set them up
   */

   ext_inptr = inptr->ext_ptr;
   ext_outptr = outptr->ext_ptr;

#endif /* } */

/*****************************************************************
 *
 * main body
 *
 *****************************************************************/

   fault_active = PSEUDO(read your hardware to determine bus error);

   if ((flagptr->clr_be) && (fault_active))
   {

      /*
         we are assumming that if we get a fault, all bets are off
         so clear out all interrupting cases which include
         all faults (including bus error) and DMA complete.

         in past products, the kernel knows that a bus error
```

```
      means reset, so lets not disappoint him.
   */

   PSEUDO(write your hardware to clear bus error)

   PSEUDO(write your hardware to clear dma complete)

   /*
      save state of interrupt enables
   */

   if (flagptr->retain_ie)
   {
      int_enables = PSEUDO(read your hardware to save current int enables)
   }

   /*
      Asked to clear the bus error status ...
      so lets do that promised RESET
   */

   PSEUDO(reset your hardware)

   /*
      restore state of interrupt enables and fault enables
   */

   if (flagptr->retain_ie)
   {
      PSEUDO(restore the int enables that were present before reset)
   }
}

/*
   Read the real thing again (affected by RESET)
*/

fault_active = PSEUDO(read your hardware to determine bus error);

outptr->be = fault_active;

/*
   we don't do anything if there is a pending bus error
*/

if (outptr->be == 0) {
```

```
/*
   if hardware does not use PA RISC architected EIM
   just ignore requests to read and write EIM
*/

/*
   if we are asked to write int enables then do so
   make sure to honor the no_chg_gie flag
*/

if (flagptr->write_int_enb_mask)
{
   if (flagptr->no_chg_gie)
      global_enable = PSEUDO(read your hardware to determine
         if the global enable is set/clear)

   int_emask.all = ext_inptr->int_enable_mask;

   if (int_emask.bits.bdc)
      PSEUDO(write your hardware to enable DMA complete interrupt)

   if (flagptr->no_chg_gie)
      PSEUDO(write your hardware global enable) = global_enable;

}

/*
   Now set the global int mask to really enable
   here is where we write global HIE if is changing,
   if it didn't change we caught that above.
*/

if (!flagptr->no_chg_gie)
{
   if (flagptr->global_int_enable)
   {
      PSEUDO(write your hardware to enable global interrupts)
      global_enable = 1;
   }
   else
   {
      /*
         must be disable
      */
```

```
            PSEUDO(write your hardware to disable global interrupts)
            global_enable = 0;
        }
    }

    /*
        if we are asked to read int enables then do so

        this should return the bdc flag if DMA complete interrupts
        are enabled
    */

    if (flagptr->read_int_enb_mask)
    {
        int_emask.all = 0;

        if PSEUDO(read hardware has DMA complete enabled)
            int_emask.bits.bdc = 1;

        ext_outptr->int_enable_mask = int_emask.all;
    }

    /*
        now set global enable in outptr
    */

    if PSEUDO(your hardware has global int enable set)
        global_enable = 1;
    else
        global_enable = 0;

    outptr->ints_enabled = global_enable;

    /*
        we have all ready done error checking to make sure
        both save and restore are not both set
    */

    /*
        begin int cycle will actually do a status read to force
        the beginning of the interrupt cycle

        NOTE: IF YOUR HARDWARE DOES NOT SUPPORT PENDING
INTERRUPTS,
              YOU  MAY MISS SOME !!!
    */
```

```
if (flagptr->begin_int_cycle)
{

   /*
      determine if there are pending interrupts
   */

   PSEUDO(read your hardware status)

   /*
      look at all ints, not just global int
      kernel will handle it properly
   */

   if PSEUDO(your hardware status indicates INTERRUPT REQUEST)
   {
      outptr->int_pend = 1;
      int_pend.all = 0;       /* set all to not pending */
      if PSEUDO(your hardware status indicates DMA Complete Interrupt)
         int_pend.bits.bdc = 1;

      ext_outptr->int_state = int_pend.all;
   }
   else
   {
      outptr->int_pend = 0;
      ext_outptr->int_state =  0;
   }
}

/*
   if we are asked to clear ints, then clear all in int_clear_mask
*/
if (flagptr->clr_int)
{
   /*
      now clear ints passed in mask
   */

   int_clear_mask.all  = (uint32_t) ext_inptr->int_clear_mask;

   /*
      clear the ints indicatied in the mask passed in.
      BUT DON'T WRITE HIC, THAT WILL BE DONE IN end_int_cycle
   */
```

```
        if (int_clear_mask.bits.bdc)
           PSEUDO(clear your hardware DMA complete bit)
        else
           PSEUDO(don't clear your hardware DMA complete bit)

        /*
           and return new status of pending interrupts
        */

        PSEUDO(read your hardware status)

        /*
           look at all ints, not just global int
           kernel will handle it properly
        */

        if PSEUDO(your hardware status indicates INTERRUPT REQUEST)
        {
           outptr->int_pend = 1;
           int_pend.all = 0;       /* set all to not pending */
           if PSEUDO(your hardware status indicates DMA Complete Interrupt)
              int_pend.bits.bdc = 1;

           ext_outptr->int_state = int_pend.all;
        }
        else
        {
           outptr->int_pend = 0;
           ext_outptr->int_state = 0;
        }
     }

     /*
        now we need to end the interrupt cycle
     */
     if (flagptr->end_int_cycle)
     {
        PSEUDO(cause your hardware to end the interrupt cycle)
     }

  }

/*****************************************************************
 *
 * done
```

```
 *
 ************************************************************/

   NORMAL_RETURN;

/************************************************************
 *
 * std exit
 *
 ************************************************************/

   EXIT_LABEL:

   EXIT_FROM_STI;
}
```

6.7 Multiple Routines

The STI code routines stored in the STI ROM are pointed to by two Device Data Fields: OFF_UX and OFF_ALT_CODE. These pointers designate the location of two sets of STI routines, on set written for the 32 bit PA-RISC and one set written for the 64 bit PA-RISC versions of HP-UX. In previous STI revisions, these sets of routines were written for different CPU architectures, such as the 68000 and PA-RISC, rather than similar architectures. And other revisions allowed for completely different operating systems, NT and HP-UX, using the same processor, the PCX-L PA-RISC processor. Note that STI with NT routines have never been released on products from our Lab and thus never really supported.

Current implementations use the two code pointers for 32 bit and 64 bit PA-RISC HP-UX only. If alternate routines are present, and the OFF_ALT_CODE type is zero, then more than likely you have an older version hardware that has 68000 code.

6.8 Frequency Reference

The extended global config structure has a field called freq_ref that was added in the 8.04 revision STI. This was added to support graphics devices that use a clock dependent upon the bus clock, rather than an independent on-board clock. The IO clock frequency is provided to STI code routines through this field in the STI data structures. There is no mechanism to determine if particular graphics hardware architecture required this so you should always provide the bus frequency.

This was mainly used on a GSC based graphics product that was intended to be embedded on the SPU board. Notably, the 712, 715 and 725 product family used this graphics card, but a plug in card was later made with the same graphics chip.

6.9 Extra Memory

STI does not support any type of global data, thus STI routines cannot share data between themselves or with the boot ROM and kernel, revision 8.04 STI allows extra memory to be set aside for saving information to be shared by the STI routines, boot ROM and kernel. This is especially useful for products that don't have offscreen space to use for storing information. This memory is pointed to by the Device Data field off_sti_mem_req. The extended global config data structure also supports this extra memory by having the field sti_mem_addr. Note, however, that this extra memory is not copied by the kernel from the boot ROM usage.

6.10 Implicit locking vs. explicit locking

The explicit locking, or fast lock, scheme used for graphics locking has been successfully used by HP for years, but with the introduction of new APIs, notably OpenGL, this was changed.

OpenGL is a vertex API rather than a primitive API. This means that each component of a primitive, for instance, color, normal, texture coordinate and vertex, are specified with a separate procedure call. Using the current scheme would require a fast lock to be acquired and released for each call.

What is required is a zero overhead locking mechanism and this is called implicit locking.

Another feature of this design is the hardware flow control. Previously, graphics processes checked the pipeline depth each time a write was done to the pipe. However, the newer buses on HP systems make reading via programmed IO a slow operation, the newer buses are optimized for write performance. In order to improve performance, new graphics hardware will interrupt when the pipeline depth exceeds a high water mark. The graphics driver will revoke access permission to the device until the pipeline depth goes below a low water mark. This will allow graphics process to not have to worry about overflowing the pipe and reduces the number of bus transactions to the graphics device.

Another feature of this design is the support for user process initiated DMA. This is also being done due to the slow performance of programmed IO reads from the graphics device. Currently, DMA will only be used for reading back from the graphics device, never to write to it.

6.11 CFB

STI has a pointer to the CFB (color frame buffer) X11 driver that is in the ROM. This is a minimal, and very generic, X11 driver that will allow the system to use X11 to install

the fully accelerated X11 driver. This is mainly used for new installations of the operating system.

STI does not use the CFB driver but will include it as part of the ROM image. The STI development tool set has a mechanism to add the CFB executable to the STI image and create an architected DD struct pointer to it so that the X11 server can find the driver.

6.12 VGA Support

STI has been written to work with some specific VGA hardware based on the PCI bus but on a case by case basis. There are numerous obstacles present that keep a generic implementation of VGA from working.

On older PA-RISC platforms, there was support for the EISA bus, which allowed an ISA based VGA card to be accessed, but older boot ROM's were not written to support graphics on the EISA bus.

On newer PA-RISC platforms, with PCI bus, the kernel currently does not support access to IO space registers for VGA, so a VGA that implements memory mapped IO must be used. This is expected to be remedied with a patch to the current versions of HP-UX.

The basic limitation of newer hardware is that all memory and I/O spaces must be PCI remappable.

6.13 VM STI

Boot ROM limitations of 10k bytes for executable STI routines are inherent to the PDC/IODC architecture of the PA-RISC platform. Basic PDC/IODC architecture limits a single IODC to 16k bytes. The console IODC contains both the keyboard IODC and the STI IODC. The keyboard IODC is given 6k bytes and the remaining 10k bytes are given to STI routines.

There have been products released that require more than 10k bytes, in particular for the initialization code (init_graph), and the solution to this was to have the STI code itself use an overlay technique to manage its 10k code space. This was called VM STI, for Virtual Memory STI. The STI routines would include their own virtual memory manager and page in and out various pieces of the STI code as required. See the tools documentation for further details.

6.14 Multiple Monitors

Revision 8.04 STI supports multiple monitors by including in the architecture two monitor related Device Data fields, monitor related fields in the STI data structures, and a new data structure used as a monitor table descriptor. The Device Data fields re

OFF_NUM_MONS and OFF_MON_TBL_ADDR, which are pointers to the number of monitors supported by STI and the monitor table, respectively. The monitor table is the table displayed during boot admin mode from which the user can choose the monitor type to have configured with the system. The data structure mon_tbl_desc contains the monitor information used in this monitor table. For STI code to configure the monitor type for the system, the fields within the data structures are used. These fields are curr_mon in the extended global config structure, cmt_chg in the init_flags structure, and config_mon_type in the init_inptr_ext structure (the extended init_graph structure). Curr_mon keeps track of the current monitor type being used, config_mon_type is the monitor type which is configured with the system, and cmt_chg signals that the current monitor type is to be changed.

STI code usually accomplishes the monitor configuration, which requires the monitor timing information. For current STI projects, this timing information is kept in the user_data space, which is explained below. Thus, when the monitor is to be configured, the STI code refers to the config_mon_type structure field to determine which monitor to configure and then uses the timing information, in the user_data space, corresponding to this monitor type to complete the configuration. Note that when the user chooses the monitor type from the boot admin mode, this type valued is returned to the boot ROM which then puts the value into the config_mon_type field in the data structure and calls STI to initialize the system with this monitor type.

6.14.1 User Data Space

User data is generic space, located within the STI ROM that is available for STI use only. This space is declared by two Device Data fields: off_user_data_addr and off_user_data_size, which, respectively, point to the space and indicate the size of this space. This space can be used for any purpose. As mentioned previously, current STI projects use this space for storing monitor timing data.

6.15 Early Console (aka Friendly Boot)

Revision 8.04 STI, and later, support early console, which is a temporary console display (in the form of a login window) that is shown onscreen during the bootup process until the startup of the X-Server, which displays its own login window through which the user actually logs in.

This early console display provides an alternative display during bootup other than the usual raw-mode console messages. Of course, the user can choose to see the raw-mode messages by toggling off the early console display, which would allow the console messages to be seen, and vice versa. STT support this toggle capability. The display process of the early console is known as “friendly boot” mode, as opposed to raw mode, in which the console messages are displayed.

This early console display is achieved via the nontext image planes. The console message text, on the other hand, is display via the text image planes. STI controls which planes are text and nontext. Although the boot ROM actually provides the early console display, which is put onto the screen via STI routine calls by the boot ROM, STI must have the text and nontext planes organized and the color map set with the appropriate colors for the early console display.

For setting up the color map for early console, the STI architecture included an additional STI routine, set_cm_entry, which sets specific colormap entries to specific colors, both provided by the calling routine (boot ROM). For aid in whether the text or nontext planes were to be used for onscreen displays, a non_text field (flag) was added to both the font_flags structure and the blkmv_flags structure. Thus, the STI caller, either the boot ROM or the kernel, would set the non_text flag for screen writes meant for the early console display. Otherwise, with non_text flags not set, the STI routines would know to display text by writing to the text planes. Note that this control of the text and nontext planes enables the toggle capability between the friendly boot mode and raw mode: early console is display when nontext planes are on, and raw mode console messages are display when text planes are on.

6.16 Multiple Fonts

While STI always supported multiple font families, for instance, ROMAN and KATAKANA, they were always based on the same size, for example a 10x20 character glyph.

With revision 8.06 STI, multiple font sizes were added. This was useful for products that had different resolutions available. When supporting resolutions from 640x480 all the way up to 1600x1200, you do not want to always use the same character cell size.

Monitor timings with 640x480 look better with a 5x9 font than they do with a 10x20 font. Likewise, it is hard to read a 5x9 font on a 1600x1200 resolution.

The specific font of the multiple font selection to be used by STI is indicated with the font index stored in the index field of the mon_tbl_desc structure.

6.17 Termcap

Older versions of the HP-UX operating system required a unique termcap database file so the ITE would be adjusted for the proper control sequences. The 10.20 version of HP-UX (and later) now determine this almost automatically.

Here is an example that could be defined in a file called ite.tic. Run the tic program against this to install it into the /usr/lib/terminfo/I directory. This is automatically included in the HP-UX distribution, but you may need to define a new entry if you create a different resolution.

The “tic” program is a terminfo compiler. It will translate terminfo files from their source format into a compiled format that is usable by the Kernel ITE (internal terminal emulator). See the untic, tic, curses and terminfo man pages for more details.

```
#
# for STI based graphics devices, base settings for ITE
#
ITE_STI|Internal terminal Emulator -- STI based subset,
        am, xhp, da, db, mir,
        cols#128, lines#49, lm#0, xmc#0, nlab#8, lh#2, lw#8,
        cbt=\Ei, bel=^G, cr=\r, tbc=\E3,
        clear=\E&a0y0C\EJ, el=\EK, ed=\EJ, hpa=\E&a%p1%dC,
        cup=\E&a%p1%dy%p2%dC, cud1=\EB, cub1=\b, cuf1=\EC,
        cuu1=\EA, dch1=\EP, dl1=\EM, smir=\EQ,
        smso=\E&dB, smul=\E&dD, sgr0=\E&d@, rmir=\ER,
        rmso=\E&d@, rmul=\E&d@, il1=\EL, kbs=\b,
        ktbc=\E3, kclr=\EJ, kctab=\E2, kdch1=\EP,
        kdl1=\EM, kcud1=\EB, krmir=\ER, kel=\EK,
        ked=\EJ, kf1=\Ep, kf2=\Eq, kf3=\Er,
        kf4=\Es, kf5=\Et, kf6=\Eu, kf7=\Ev,
        kf8=\Ew, khome=\Eh, kich1=\EQ, kil1=\EL,
        kcub1=\ED, kll=\EF, knp=\EU, kpp=\EV,
        kcuf1=\EC, kind=\ES, kri=\ET, khts=\E1,
        kcuu1=\EA, rmkx=\E&s0A, smkx=\E&s1A,
        pfkey=\E&f%p1%dk%p2%l%dL%p2%s,
        pfloc=\E&f1a%p1%dk%p2%l%dL%p2%s,
        pfx=\E&f2a%p1%dk%p2%l%dL%p2%s, vpa=\E&a%p1%dY, ind=\n,
        sgr=\E&d%?%p7%t%'s'%c%;%p1%p3%|%p6%|%{2}%*%p2%{4}%*%+%p4%
+%p5%{8}%*%+%'@'%+%c%?%p9%t%'^N'%c%e%'^O'%c%;, hts=\E1, ht=\t,
pln=\E&f%p1%dk%p2%l%dd0L%p2%s,
        smln=\E&jB, rmln=\E&j@,


#
# for STI based graphics devices, 1600x1200 resolution screen and 10x20 font
#
ITE_x160y58|ite sti --  1600x1200 resolution with 10x20 font,
        cols#160, lines#58, use=ITE_STI,

#
# for STI based graphics devices, 1200x1600 resolution screen and 10x20 font
#
ITE_x120y78|ite sti --  1200x1600 resolution with 10x20 font,
        cols#120, lines#78, use=ITE_STI,
```

```
#
# for STI based graphics devices, 1280x1024 resolution screen and 10x20 font
#
ITE_x128y49|ite sti --  1280x1024 resolution with 10x20 font,
        cols#128, lines#49, use=ITE_STI,

#
# for STI based graphics devices, 1280x1024 resolution screen and 8x16 font
#
ITE_x160y62|ite sti --  1280x1024 resolution with 8x16 font,
        cols#160, lines#62, use=ITE_STI,

#
# for STI based graphics devices, 1024x768 resolution screen and 8x16 font
#
ITE_x128y46|ite sti --  1024x768 resolution with 8x16 font,
        cols#128, lines#46, use=ITE_STI,

#
# for STI based graphics devices, 800x600 resolution screen and 6x13 font
#
ITE_x133y44|ite sti --  800x600 resolution with 6x13 font,
        cols#133, lines#44, use=ITE_STI,

#
# for STI based graphics devices, 640x480 resolution screen and 6x13 font
#
ITE_x106y34|ite sti --  640x480 resolution with 6x13 font,
        cols#106, lines#34, use=ITE_STI,
```

6.18 IODC SV - firmware info

System firmware uses 0x85 for word mode STI and 0x77 for byte mode STI as the IODC sversion field. You will only care about this if you are writing PDC/IODC for new systems or PDC/IODC interfaces for testing.

7 Appendix

7.1 64 bit ID

Each different graphics product using the STI is required to have a unique ID to allow it to be uniquely identified. The idea is similar to the UUID used in DCE and the GUID used by Microsoft.

Contact Hewlett-Packard for a complete list of current assignments or to create a new assignment.

7.2 Compatibility between revisions

In general, completely proper operation should only be expected when the kernel and boot ROM code are written to the same spec revision level as the STI ROM code. Unfortunately, after products have been released we will begin to encounter situations with customer's having systems that contain components with code written to different STI spec revisions. Because of this, major effort will be taken to insure that incompatibility problems are minimized due to spec changes. Code writers, for both calling routines and ROM code, should also try to make the spec revision changes in their code in a manner designed to minimize the effects of incompatibilities.
.

7.3 crc algo

Below is a sample C program that will perform the necessary CRC on the STI ROM contents.

```
#define OFF_LAST_ADDR  (0x0053/4)

/*****************************************************************
 *
 * NAME
 *        check_crc
 *
 * PURPOSE
 *        Calculates the CRC for the ROM to make sure it isn't
 *        corrupted.  It returns 0 if the CRC is ok, and -1 if
 *        it doesn't check out correctly.
 *
 *****************************************************************/
     check_crc(rom)
```

```
        int *rom;
{
        int *c,*romend;
        unsigned short code, poly, accum;
        int i,j;

        /* get size of rom from last_addr entry */
        i = ((rom[OFF_LAST_ADDR+0] & 0xff)<< 24)
                | ((rom[OFF_LAST_ADDR+1] & 0xff)<< 16)
           | ((rom[OFF_LAST_ADDR+2] & 0xff)<< 8)
           |  (rom[OFF_LAST_ADDR+3] & 0xff);
        romend = rom + (i/sizeof(int));

        /* calculate CRC of ROM */
        accum = 0;
        code = 0;
        poly = 0x8408;
        for (c = rom,j=0; c <= romend; c++,j++) {
                accum = (accum << 8) | (*c & 0xff);
                if (j & 1) {
                   accum ^= code;
                   for (i=0; i<16; i++) {
                           /* do a left rotate */
                           if (accum & 0x8000) {
                                   accum = (accum << 1) | 0x0001;
                                   accum ^= poly;
                           }
                           else
                                   accum <<= 1;
                   }
                   code = accum;
                }
        }
        if (code != 0)
                return(-1);
        else
                return(0);
}
```

7.4 header files

7.4.1 sti.h

```
/* sti.h (revision 8.0d) */

#include "inttypes.h"
#include "framebuf.h"

#ifndef _STI_GLOBAL_STI_H /* { */
#define _STI_GLOBAL_STI_H

/* sti module return parameters */
#define PASS           0
#define FAIL          -1
#define NOT_RDY        1

/* other defs */
#define REGION_MAX         8
#define MONITOR_MAX        256
#define DEV_NAME_LENGTH    32

/*

Bit decode for BUS_SUPPORT_PCI_DUAL_DECODE
   0 - Dual Decoder
   1 - Shared Decoder (1 decoder only)

Bit decode for BUS_SUPPORT_PCI_EROM_MMAP
   0 - EROM memory mapped
   1 - EROM not memory mapped

*/

#define BUS_SUPPORT_GSCINTL        0x01
#define BUS_SUPPORT_GSC15X         0x02
#define BUS_SUPPORT_GSC2X          0x04
#define BUS_SUPPORT_PCI_IOEIM      0x08
#define BUS_SUPPORT_IMPLICIT_LOCK  0x10
#define BUS_SUPPORT_PCI_DUAL_DECODE    0x20
#define BUS_SUPPORT_PCI_EROM_MMAP      0x40
#define BUS_SUPPORT_PCI_STD_INT        0x80

#define EXT_BUS_SUPPORT_DMA              0x01
```

```
#define EXT_BUS_SUPPORT_IMPLICIT_LOCK_PIO   0x02

#define ALT_CODE_TYPE_UNKNOWN          0x00
#define ALT_CODE_TYPE_PA_RISC_64       0x01

/* color defs */

#define BLACK       0
#define WHITE       1
#define RED         2
#define YELLOW      3
#define GREEN       4
#define CYAN        5
#define BLUE        6
#define MAGENTA     7

/* offsets for various components of STI rom */
/* these are the offsets used for word mode roms */
#define WM_OFF_TYPE            (0x0000)
#define WM_OFF_NUM_MONS        (0x0005)
#define WM_OFF_REVNO           (0x0006)
#define WM_OFF_GRAPHICS_ID     (0x0008)
#define WM_OFF_FONT_START      (0x0010)
#define WM_OFF_STATESIZE       (0x0014)
#define WM_OFF_LAST_ADDR       (0x0018)
#define WM_OFF_REGION_LIST     (0x001c)
#define WM_OFF_REENTSIZE       (0x0020)
#define WM_OFF_MAXTIME         (0x0022)

#define WM_OFF_MON_TBL_ADDR    (0x0024)
#define WM_OFF_USER_DATA_ADDR  (0x0028)
#define WM_OFF_STI_MEM_REQ     (0x002c)
#define WM_OFF_USER_DATA_SIZE  (0x0030)
#define WM_OFF_POWER           (0x0034)
#define WM_OFF_BUS_SUPPORT     (0x0036)
#define WM_OFF_EXT_BUS_SUPPORT (0x0037)

#define WM_OFF_ALT_CODE_TYPE   (0x0038)
#define WM_OFF_EXT_DD_STRUCT   (0x0039)
#define WM_OFF_CFB_ADDR        (0x003c)

#define WM_OFF_UX              (0x00100/4)
#define WM_OFF_PA_RISC         (0x00100/4)
#define WM_OFF_NT              (0x00200/4)
#define WM_OFF_680X0           (0x00200/4)
#define WM_OFF_ALT_CODE        (0x00200/4)
```

```
/* offsets for various components of STI rom */
/* these are the offsets used for byte mode roms */
#define BM_OFF_TYPE        (0x0003)
#define BM_OFF_NUM_MONS    (0x0007)
#define BM_OFF_REVNO       (0x000b)
#define BM_OFF_GRAPHICS_ID (0x0013)
#define BM_OFF_FONT_START  (0x0033)
#define BM_OFF_STATESIZE   (0x0043)
#define BM_OFF_LAST_ADDR   (0x0053)
#define BM_OFF_REGION_LIST (0x0063)
#define BM_OFF_REENTSIZE   (0x0073)
#define BM_OFF_MAXTIME     (0x007b)

#define BM_OFF_MON_TBL_ADDR    (0x0083)
#define BM_OFF_USER_DATA_ADDR  (0x0093)
#define BM_OFF_STI_MEM_REQ     (0x00a3)
#define BM_OFF_USER_DATA_SIZE  (0x00b3)
#define BM_OFF_POWER           (0x00c3)
#define BM_OFF_BUS_SUPPORT     (0x00cb)
#define BM_OFF_EXT_BUS_SUPPORT (0x00cf)

#define BM_OFF_ALT_CODE_TYPE   (0x00d3)
#define BM_OFF_EXT_DD_STRUCT   (0x00d7)
#define BM_OFF_CFB_ADDR        (0x00e3)

#define BM_OFF_UX          (0x00103)
#define BM_OFF_PA_RISC     (0x00103)
#define BM_OFF_NT          (0x00203)
#define BM_OFF_680X0       (0x00203)
#define BM_OFF_ALT_CODE    (0x00203)

/* these are here for kernel ONLY */
#define OFF_TYPE        (0x0003/4)
#define OFF_NUM_MONS    (0x0007/4)
#define OFF_REVNO       (0x000b/4)
#define OFF_GRAPHICS_ID (0x0013/4)
#define OFF_FONT_START  (0x0033/4)
#define OFF_STATESIZE   (0x0043/4)
#define OFF_LAST_ADDR   (0x0053/4)
#define OFF_REGION_LIST (0x0063/4)
#define OFF_REENTSIZE   (0x0073/4)
#define OFF_MAXTIME     (0x007b/4)

#define OFF_MON_TBL_ADDR   (0x0083/4)
#define OFF_USER_DATA_ADDR (0x0093/4)
```

```
#define OFF_STI_MEM_REQ     (0x00a3/4)
#define OFF_USER_DATA_SIZE  (0x00b3/4)
#define OFF_POWER           (0x00c3/4)
#define OFF_BUS_SUPPORT     (0x00cb/4)
#define OFF_EXT_BUS_SUPPORT (0x00cf/4)

#define OFF_ALT_CODE_TYPE   (0x00d3/4)
#define OFF_EXT_DD_STRUCT   (0x00d7/4)
#define OFF_CFB_ADDR        (0x00e3/4)

#define OFF_UX              (0x00103/4)
#define OFF_PA_RISC         (0x00103/4)
#define OFF_NT              (0x00203/4)
#define OFF_680X0           (0x00203/4)
#define OFF_ALT_CODE        (0x00203/4)

/*****************************************************************
 *
 * global config structures
 *
 *****************************************************************/

typedef struct {
   uint8_t    curr_mon;            /* current monitor configured */
   uint8_t    friendly_boot;       /* in friendly boot mode */
   int16_t    power;               /* power calculation */
   int32_t    freq_ref;            /* frequency refrence */
   int32_t    *sti_mem_addr;       /* pointer to global sti memory */
   int32_t    *future_ptr;         /* pointer to future data */
} glob_cfg_ext;

typedef struct {
   int32_t    text_planes;         /* number of planes used for text */
   int16_t    onscreen_x;          /* screen width in pixels */
   int16_t    onscreen_y;          /* screen height in pixels */
   int16_t    offscreen_x;         /* offscreen width in pixels */
   int16_t    offscreen_y;         /* offscreen height in pixels */
   int16_t    total_x;             /* frame buffer width in pixels */
   int16_t    total_y;             /* frame buffer height in pixels */
   int32_t    *region_ptrs[REGION_MAX];   /* region pointers */
   int32_t    reent_lvl;           /* storage for reentry level value */
   int32_t    *save_addr;          /* where to save or restore reentrant state */
   glob_cfg_ext *ext_ptr;          /* pointer to extended glob_cfg data structure */
} glob_cfg;

/*****************************************************************
```

```
 *
 * region descriptor structure
 *
 ***********************************************************************/

typedef struct {
   uint32_t    offset      : 14;   /* offset in 4kbyte page */
   uint32_t    sys_only    : 1;    /* don't map to user space */
   uint32_t    cache       : 1;    /* map to data cache */
   uint32_t    btlb        : 1;    /* map to block tlb */
   uint32_t    last        : 1;    /* last region in list */
   uint32_t    length      : 14;   /* length in 4kbyte page */
} region_desc;

/***********************************************************************
 *
 * monitor table descriptor structure
 *
 ***********************************************************************/

typedef struct {
   uint32_t    x             : 12;
   uint32_t    y             : 12;
   uint32_t    hz            : 7;
   uint32_t    class_flat      : 1;
   uint32_t    class_vesa      : 1;
   uint32_t    class_grey      : 1;
   uint32_t    class_dbl       : 1;
   uint32_t    class_user      : 1;
   uint32_t    class_stereo    : 1;
   uint32_t    class_sam       : 1;
   uint32_t    pad           : 15;
   uint32_t    hz_upper        : 3;
   uint32_t    index         : 8;
} mon_tbl_desc;

/***********************************************************************
 *
 * interrupt structure
 *
 ***********************************************************************/

typedef struct {
   uint32_t    bet     : 1;    /* bus error timer */
   uint32_t    hw      : 1;    /* high water */
   uint32_t    lw      : 1;    /* low water */
```

```
   uint32_t    tm      : 1;    /* texture map */
   uint32_t    vb      : 1;    /* vertical blank */
   uint32_t    udc     : 1;    /* un-buffered dma complete */
   uint32_t    bdc     : 1;    /* buffered dma complete */
   uint32_t    udpc    : 1;    /* un-buffered priv dma complete */
   uint32_t    bdpc    : 1;    /* buffered priv dma complete */
   uint32_t    pad     : 23;
} int_desc;

/*************************************************************
 *
 * init_graph structures
 *
 *  init_graph(&init_flags,&init_inptr,&init_outptr,&glob_cfg)
 *
 *************************************************************/

typedef struct {
   uint32_t    wait           : 1;    /* should routine idle wait or not */
   uint32_t    reset          : 1;    /* hard reset the device? */
   uint32_t    text           : 1;    /* turn on text display planes? */
   uint32_t    nontext        : 1;    /* turn on non-text display planes? */
   uint32_t    clear          : 1;    /* clear text display planes? */
   uint32_t    cmap_blk       : 1;    /* non-text planes cmap black? */
   uint32_t    enable_be_timer : 1;    /* enable bus error timer */
   uint32_t    enable_be_int   : 1;    /* enable bus error timer interrupt */
   uint32_t    no_chg_tx      : 1;    /* don't change text settings */
   uint32_t    no_chg_ntx     : 1;    /* don't change non-text settings */
   uint32_t    no_chg_bet     : 1;    /* don't change berr timer settings */
   uint32_t    no_chg_bei     : 1;    /* don't change berr int settings */
   uint32_t    init_cmap_tx   : 1;    /* initialize cmap for text planes */
   uint32_t    cmt_chg        : 1;    /* change current monitor type */
   uint32_t    retain_ie      : 1;    /* don't allow reset to clear int enables */
   uint32_t    pad            : 17;   /* pad to word boundary */
   int32_t     *future_ptr;           /* pointer to future data */
} init_flags;

typedef struct {
   uint8_t     config_mon_type;    /* configure to monitor type */
   uint8_t     pad[1];             /* pad to word boundary */
   uint16_t    inflight_data;      /* inflight data possible on PCI */
   int32_t     *future_ptr;        /* pointer to future data */
} init_inptr_ext;

typedef struct {
   int32_t     text_planes;        /* number of planes to use for text */
```

```
   init_inptr_ext *ext_ptr;       /* pointer to extended init_graph inptr data structure*/
} init_inptr;

typedef struct {
   int32_t    errno;            /* error number on failure */
   int32_t    text_planes;      /* number of planes used for text */
   int32_t    *future_ptr;      /* pointer to future data */
} init_outptr;

/*****************************************************************
 *
 * state_mgmt structures
 *
 *  state_mgmt(&state_flags,&state_inptr,&state_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t   wait         : 1;  /* should routine idle wait or not */
   uint32_t   save         : 1;  /* save (1) or restore (0) state */
   uint32_t   res_disp     : 1;  /* restore all display planes */
   uint32_t   pad          : 29; /* pad to word boundary */
   int32_t    *future_ptr;       /* pointer to future data */
} state_flags;

typedef struct {
   int32_t    *save_addr;       /* where to save or restore state */
   int32_t    *future_ptr;      /* pointer to future data */
} state_inptr;

typedef struct {
   int32_t    errno;            /* error number on failure */
   int32_t    *future_ptr;      /* pointer to future data */
} state_outptr;

/*****************************************************************
 *
 * font_unpmv structures
 *
 *  font_unpmv(&font_flags,&font_inptr,&font_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t   wait         : 1;  /* should routine idle wait or not */
   uint32_t   non_text     : 1;  /* font unpack/move in non_text planes =1,text =0 */
```

```
   uint32_t    pad          : 30; /* pad to word boundary */
   int32_t     *future_ptr;       /* pointer to future data */
} font_flags;

typedef struct {
   int32_t     *font_start_addr;   /* address of font start */
   int16_t     index;             /* index into font table of character */
   uint8_t     fg_color;          /* foreground color of character */
   uint8_t     bg_color;          /* background color of character */
   int16_t     dest_x;            /* X location of character upper left */
   int16_t     dest_y;            /* Y location of character upper left */
   int32_t     *future_ptr;       /* pointer to future data */
} font_inptr;

typedef struct {
   int32_t     errno;             /* error number on failure */
   int32_t     *future_ptr;       /* pointer to future data */
} font_outptr;

/*****************************************************************
 *
 * block_move structures
 *
 *  block_move(&blkmv_flags,&blkmv_inptr,&blkmv_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t    wait         : 1;  /* should routine idle wait or not */
   uint32_t    color        : 1;  /* change color during move? */
   uint32_t    clear        : 1;  /* clear during move? */
   uint32_t    non_text     : 1;  /* block move in non_text planes =1,text =0 */
   uint32_t    pad          : 28; /* pad to word boundary */
   int32_t     *future_ptr;       /* pointer to future data */
} blkmv_flags;

typedef struct {
   uint8_t     fg_color;       /* foreground color after move */
   uint8_t     bg_color;       /* background color after move */
   int16_t     src_x;          /* source upper left pixel x location */
   int16_t     src_y;          /* source upper left pixel y location */
   int16_t     dest_x;         /* dest upper left pixel x location */
   int16_t     dest_y;         /* dest upper left pixel y location */
   int16_t     width;          /* block width in pixels */
   int16_t     height;         /* block height in pixels */
   int32_t     *future_ptr;    /* pointer to future data */
```

```
} blkmv_inptr;

typedef struct {
   int32_t     errno;          /* error number on failure */
   int32_t     *future_ptr;    /* pointer to future data */
} blkmv_outptr;

/*****************************************************************
 *
 * self_test structures
 *
 *  self_test(&test_flags,&test_inptr,&test_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t    wait         : 1;  /* should routine idle wait or not */
   uint32_t    ext_test     : 1;  /* perform extended self test */
   uint32_t    pad          : 30; /* pad to word boundary */
   int32_t     *future_ptr;       /* pointer to future data */
} test_flags;

typedef struct {
   int32_t     *future_ptr;       /* pointer to future data */
} test_inptr;

typedef struct {
   int32_t     errno;             /* error number on failure */
   int32_t     result;            /* result of the self test */
   int32_t     *future_ptr;       /* pointer to future data */
} test_outptr;

/*****************************************************************
 *
 * excep_hdlr structures
 *
 *  excep_hdlr(&excep_flags,&excep_inptr,&excep_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t    wait             : 1;    /* should routine idle wait or not */
   uint32_t    clr_int          : 1;    /* should routine clr int or not */
   uint32_t    clr_be           : 1;    /* should routine clr be stat or not */
   uint32_t    save_int         : 1;    /* should int state be saved or not */
   uint32_t    restore_int      : 1;    /* should int state be restored or not*/
```

```
    uint32_t    write_eim          : 1;    /* write eim with inptr_ext  */
    uint32_t    read_eim           : 1;    /* read eim to outptr_ext */
    uint32_t    global_int_enable   : 1;
    uint32_t    no_chg_gie         : 1;
    uint32_t    write_int_enb_mask  : 1;
    uint32_t    read_int_enb_mask   : 1;
    uint32_t    begin_int_cycle     : 1;
    uint32_t    end_int_cycle      : 1;
    uint32_t    retain_ie          : 1;    /* don't allow reset to clear int enables */
    uint32_t    pad                : 18;   /* pad to word boundary */
    int32_t     *future_ptr;               /* pointer to future data */
} excep_flags;

typedef struct {
    uint32_t    eim_addr;
    uint32_t    eim_data;
    uint32_t    int_enable_mask;        /* for setting int enables */
    uint32_t    int_clear_mask;         /* for clearing pending ints */
    int32_t     *future_ptr;
} excep_inptr_ext;

typedef struct {
    int32_t     *save_addr;            /* where to save or restore int state */
    excep_inptr_ext  *ext_ptr;
} excep_inptr;

typedef struct {
    uint32_t    eim_addr;
    uint32_t    eim_data;
    uint32_t    int_enable_mask;        /* for readback of enables */
    uint32_t    int_state;
    int32_t     *future_ptr;
} excep_outptr_ext;

typedef struct {
    int32_t     errno;                 /* error number on failure */
    uint32_t    be             : 1;    /* was be intercepted or not */
    uint32_t    int_pend       : 1;    /* is there an existing int or not */
    uint32_t    ints_enabled    : 1;    /* is global enable set ? */
    uint32_t    pad            : 29;   /* pad to word boundary */
    excep_outptr_ext    *ext_ptr;
} excep_outptr;

/*****************************************************************
 *
 * inq_conf structures
```

```
 *
 *  inq_conf(&conf_flags,&conf_inptr,&conf_outptr,&glob_cfg)
 *
 ***************************************************************/

typedef struct {
   uint32_t    wait         : 1;  /* should routine idle wait or not */
   uint32_t    pad          : 31; /* pad to word boundary */
   int32_t     *future_ptr;       /* pointer to future data */
} conf_flags;

typedef struct {
   int32_t     *future_ptr;       /* pointer to future data */
} conf_inptr;

typedef struct {
   uint32_t    crt_config[3];
   uint32_t    crt_hdw[3];
   int32_t     *future_ptr;
} conf_outptr_ext;

typedef struct {
   int32_t     errno;             /* error number on failure */
   int16_t     onscreen_x;        /* screen width in pixels */
   int16_t     onscreen_y;        /* screen height in pixels */
   int16_t     offscreen_x;       /* offscreen width in pixels */
   int16_t     offscreen_y;       /* offscreen height in pixels */
   int16_t     total_x;           /* frame buffer width in pixels */
   int16_t     total_y;           /* frame buffer height in pixels */
   int32_t     bits_per_pixel;    /* bits/pixel device has configured */
   int32_t     bits_used;         /* bits which can be accessed */
   int32_t     planes;            /* number of fb planes in system */
   uint8_t     dev_name[DEV_NAME_LENGTH];  /* null terminated product name */
   uint32_t    attributes;        /* flags denoting attributes */
   conf_outptr_ext *ext_ptr;      /* pointer to future data */
} conf_outptr;

/***************************************************************
 *
 * set_colormap_entry structures
 *
 *  set_cm_entry(&setcm_flags,&setcm_inptr,&setcm_outptr,&glob_cfg)
 *
 ***************************************************************/

typedef struct {
```

```
   uint32_t    wait         : 1;  /* should routine idle wait or not */
   uint32_t    pad          : 31; /* pad to word boundary */
   int32_t     *future_ptr;       /* pointer to future data */
} setcm_flags;

typedef struct {
   int32_t     entry;             /* entry number */
   uint32_t    value;              /* entry value */
   int32_t     *future_ptr;       /* pointer to future data */
} setcm_inptr;

typedef struct {
   int32_t     errno;             /* error number on failure */
   int32_t     *future_ptr;       /* pointer to future data */
} setcm_outptr;

/*****************************************************************
 *
 * dma_ctrl structures
 *
 *  dma_ctrl(&dma_flags,&dma_inptr,&dma_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t    wait           : 1;    /* should routine idle wait or not */
   uint32_t    priv           : 1;    /* priv or non-priv */
   uint32_t    disable         : 1;
   uint32_t    buff           : 1;    /* buffered or un-buffered */
   uint32_t    marker          : 1;    /* write a marker */
   uint32_t    abort          : 1;    /* abort dma xfer */
   uint32_t    pad            : 26;   /* pad to word boundary */
   int32_t     *future_ptr;           /* pointer to future data */
} dma_flags;

typedef struct {
   int32_t     phys_addr_upper;
   int32_t     phys_addr_lower;
   int32_t     size;
   int32_t     marker_data;
   int32_t     marker_offset;
   int32_t     *future_ptr;           /* pointer to future data */
} dma_inptr;

typedef struct {
   int32_t     errno;                 /* error number on failure */
```

```
   int32_t    *future_ptr;          /* pointer to future data */
} dma_outptr;

/*****************************************************************
 *
 * flow_ctrl structures
 *
 *  flow_ctrl(&flow_flags,&flow_inptr,&flow_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t   wait          : 1;   /* should routine idle wait or not */
   uint32_t   chk_high_water  : 1;
   uint32_t   write_hw_count  : 1;
   uint32_t   write_lw_count  : 1;
   uint32_t   no_chg_cse      : 1;
   uint32_t   cs_enable       : 1;
   uint32_t   cs_write_fine   : 1;
   uint32_t   cs_write_coarse : 1;
   uint32_t   cs_write_fifo   : 1;
   uint32_t   pad           : 23;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_flags;

typedef struct {
   int32_t    retry_count;
   int32_t    buffer_zone;
   int32_t    high_water_count;
   int32_t    low_water_count;
   int32_t    cs_fine_val;
   int32_t    cs_coarse_val;
   int32_t    cs_fifo_count;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_inptr;

typedef struct {
   int32_t    errno;               /* error number on failure */
   int32_t    retry_result;
   int32_t    fifo_size;
   int32_t    *future_ptr;          /* pointer to future data */
} flow_outptr;

/*****************************************************************
 *
 * user_timing structures
```

```
 *
 *  user_timing(&timing_flags,&timing_inptr,&timing_outptr,&glob_cfg)
 *
 ****************************************************************/

typedef struct {
   uint32_t    wait          : 1;    /* should routine idle wait or not */
   uint32_t    kbuf_size     : 1;    /* this call has kbuf_size */
   uint32_t    pad           : 30;
   int32_t     *future_ptr;          /* pointer to future data */
} timing_flags;

typedef struct {
   int32_t     *data;
   int32_t     *kbuf;
   int32_t     *future_ptr;          /* pointer to future data */
} timing_inptr;

typedef struct {
   int32_t     errno;               /* error number on failure */
   int32_t     kbuf_size;           /* size we are requesting from kernel */
                                    /* in bytes */
   int32_t     *future_ptr;          /* pointer to future data */
} timing_outptr;

/****************************************************************
 *
 * process_mgr structures
 *
 *  process_mgr(&process_flags,&process_inptr,&process_outptr,&glob_cfg)
 *
 ****************************************************************/

enum OPCODE {
      cleanup          = 0,
      buff_access_ctrl    = 1
   };

typedef struct {
   uint32_t    wait          : 1;    /* should routine idle wait or not */
   enum OPCODE opcode        : 4;
   uint32_t    critical      : 1;
   uint32_t    buff          : 1;
   uint32_t    ignore_buff   : 1;
   uint32_t    pad           : 24;
   int32_t     *future_ptr;          /* pointer to future data */
```

```
} process_flags;

typedef struct {
   int32_t    reserved[4];
   int32_t    *future_ptr;          /* pointer to future data */
} process_inptr;

typedef struct {
   int32_t    errno;               /* error number on failure */
   int32_t    *future_ptr;          /* pointer to future data */
} process_outptr;

/*****************************************************************
 *
 * sti_util structures
 *
 *  sti_util(&util_flags,&util_inptr,&util_outptr,&glob_cfg)
 *
 *****************************************************************/

typedef struct {
   uint32_t   root_user   : 1;    /* ioctl(GC_STI_UTIL) called as root */
   uint32_t   pad         : 31;
   int32_t    *future_ptr;       /* pointer to future data */
} util_flags;

typedef struct {
   int32_t    in_size;           /* size of the incoming data */
   int32_t    out_size;          /* size of the outgoing data */
   uint8_t    *buffer;           /* incoming/outgoing data */
} util_inptr;

typedef struct {
   int32_t    errno;            /* error number on failure */
   int32_t    *future_ptr;       /* pointer to future data */
} util_outptr;

/*****************************************************************
 *
 * font header structure
 *
 *****************************************************************/

typedef struct {
   uint16_t   first_char_ascii;  /* ASCII code for first char */
   uint16_t   last_char_ascii;   /* ASCII code for last char  */
```

```
    uint8_t width;          /* font width in pixels      */
    uint8_t height;         /* font height in pixels     */
    uint8_t lang_type;      /* font language type        */
    uint8_t char_size;      /* bytes/char in font        */
    uint32_t    next_font;      /* offset to next font       */
    uint8_t underline_height;   /* height in pix of underline*/
    uint8_t underline_offset;   /* offset from top of char   */
    uint8_t column_mode;        /* future use            */
    uint8_t compression_type;   /* future use            */
} font_header;

#endif /* } */
```

7.4.2 errno.h - device common error codes

This section documents those error codes that are device independent. Device independent error codes must occupy the values from 0x00 to 0xff. Device dependent error codes must occupy the range from 0x100 to 0x1ff and should be documented in the hardware ERS for each graphics device.

```
/* global errno.h (revision 8.07) */

#ifndef _STI_GLOBAL_ERRNO_H /* { */
#define _STI_GLOBAL_ERRNO_H

/* these are common to all sti routines */

#define NO_ERROR              0   /* no errno indicated */
#define BAD_REENT_LVL         1   /* bad reentry level value */
#define NO_REGIONS_DEFINED    2   /* region table is not set up */
#define ILLEGAL_NUM_PLANES    3   /* more than 3 or less than 1 text planes */
#define INVALID_INDEX         4   /* bad font index */
#define INVALID_LOC           5   /* bad font location */
#define INVALID_COLOR         6   /* bad color */
#define INVALID_BLKMV_FROM_LOC  7   /* bad blkmv from location */
#define INVALID_BLKMV_TO_LOC    8   /* bad blkmv to location */
#define INVALID_BLKMV_SIZE    9   /* bad blkmv size location */
#define NO_BE_INTR            10  /* bus error interrupts not supported */
#define UNEXPECTED_BE         11  /* unexpected bus error */
#define HDW_FAILURE           12  /* unexpected hdw failure */
#define NO_GLOB_CFG_EXT       13  /* extended global config structure is missing */
#define NO_INIT_GRAPH_EXT     14  /* extended init_graph struct is missing */
#define INVALID_CM_ENTRY      15  /* invalid set colormap entry number */
```

```
#define INVALID_CM_VALUE        16  /* invalid set colormap entry value */
#define NO_RESERVED_MEMORY      17  /* requested global memory not allocated */
#define RESERVED_MEM_CORRUPTED  18  /* requested global memory is corrupt
*/
#define INVALID_NONTEXT_BLKMV   19  /* invalid non-text blockmove requested
*/
#define BAD_MONITOR_VALUE       20  /* monitor selection is out of range */
#define INVALID_EXCEP_ADDR      21  /* bad save_addr in exception handler */



#define INVALID_EXCEP_FLAGS     22  /* bad combination excep hdlr flags */
#define NO_EXCEP_HDLR_EXT           23      /* extended excep_hdlr struct is
missing */
#define NO_INQ_CONF_EXT                 24      /* extended excep_hdlr
structure is missing */
#define INVALID_REGION_PTR          25  /* region pointer is invalid */



#define INVALID_UTIL_OPCODE         26      /* bad opcode passed to sti_util */

/* really serious errors */
#define UNKNOWN_ERROR                   250
#define NO_CONFIGPTR_DEFINED        251
#define NO_FLAGPTR_DEFINED          252
#define NO_INPTR_DEFINED            253
#define NO_OUTPTR_DEFINED           254     /* no way of passing back this errno
*/


#define NO_LOCK             255 /* kernel does not honor graphics lock */

#endif /* } */
```

7.4.3 local_errno.h - device specific error codes

These values are defined by the STI developer and are unique for each STI ROM.

7.5 Tools

Several tools are available to aid developers in the generation of STI ROM's. Contact Hewlett-Packard for access to the complete tool set.

There are two primary tools used for STI development. They are the romalizer program and the fakedrive program.

7.5.1 Romalizer

The romalizer program is responsible for compiling the STI source code and generating the final ROM image product. There are a number of configuration files that assist the developer in formatting the proper ROM image.

Typically a user will use a Makefile to generate the objects that are then passed to other utility routines. These other routines will extract the TEXT section (the executable part) out of the object files and build the appropriate code.

7.5.2 Fakedrive

The fakedrive program allows a user to test the STI code as a user program under HP-UX. Test environments include the capability of debugging STI code at the source code level, as well as assembly and machine level code.

The fakedrive program can be scripted to allow for automated testing and numerous scripts are provided to test individual routines as well as complete calling sequences that are performed by both the Boot ROM and Kernel.

7.5.3 Other

There are a number of other program and utilities that are part of the STI development environment. These include programs that :

- generate both 32bit SOM as well as 64bit ELF code,
- extract the executable code sections out of .o files produced by compilers,
- verify the ROM crc,
- perform binary edits of ROM images,
- view and decode the various fields of a ROM image,
- assist in building a VM STI routine to allow a routine to be larger than the 10k byte limitation.
- build basic byte mode, word mode and pci mode ROM images.
- generate a new 64 bit ID
- disassemble the raw executable

7.6 Ordered list of changes

The original, and only, STI specification release was done with 8.02 version. There have been 11 revision since then. The current release is 8.0d.

What follows is a list of what the various modifications were for each addition to the original release.

1. 8.02

   original release

2. 8.03

   OSF - extended self test (or fast)
   restore display

3. 8.04

   global cfg
   rename glob cfg ext
   global cfg ext
   curr_mon
   friendly boot
   freq ref
   sti_mem_addr
   dd
   type - word mode
   num mons
   mon tbl
   user data
   sti mem req
   user data size
   maxtime
   nt,ux
   mon tbl desc struct
   init_inptr
   rename init inptr ext
   init_inptr_ext
   config mon type
   cmt change
   font unpmv
   non text
   block move
   non text

   set cm entry

Word Mode
Multiple Monitors
user_data sti space usage
extra memory
unix and nt
frequency reference
early console
PCX-L, GSC Bus, Romless, Friendly Boot

4. 8.05

   interrupct support
   power usage
   Birds of Prey, User Interrupts, power

5. 8.06

   multiple fonts
   monitor table descriptor strings
   PCXL-2 and PCX-U monitor desc

6. 8.07

   PCXL-2 and PCX-U GSC2x extensions

7. 8.08

   HP-UX 10.xx support for the Visualize FX product
   dma_Ctrl
   flow_ctrl
   user_timing

8. 8.09

   Additional changes for the Visualize FX product before initial release due to a rearchitecture for performance reasons.

   process_mgr

9. 8.0a

   PCX-L2 and PCX-U dual PCI EROM map mode (Visualize EG product)

10. 8.0b

HP-UX non implicit locking DMA, implemented on the Visualize FXe product.

11. 8.0c

    STI_UTIL - flashing under HP-UX and other sideband traffic

12. 8.0d

    CFB - color framebuffer