A2410 High Resolution Color Graphics Card

Hardware / Software Overview

Richard Miner   Alex Niedzwiecki
Center for Productivity Enhancement
University of Lowell

Abstract

A high resolution color graphics card, the A2410 has been developed for the Commodore Amiga computer. This graphics card is based on a Texas Instruments graphics systems processor, the TMS34010. The card couples the graphics system processor with frame buffer and program/data memory, a palette chip and DMA circuit for high speed data transfer between the graphics card and the Amiga.

Introduction

The A2410 high resolution graphics card is a separate graphics device that sits in one of the standard Amiga 100-pin expansion slots. The graphics card couples the TI Graphics System Processor (GSP) with its own local program memory, video memory, palette chip and DMA circuit. Presented here is an overview of the main functional components of the graphics card and a description of a low level application programmers interface for accessing these capabilities.

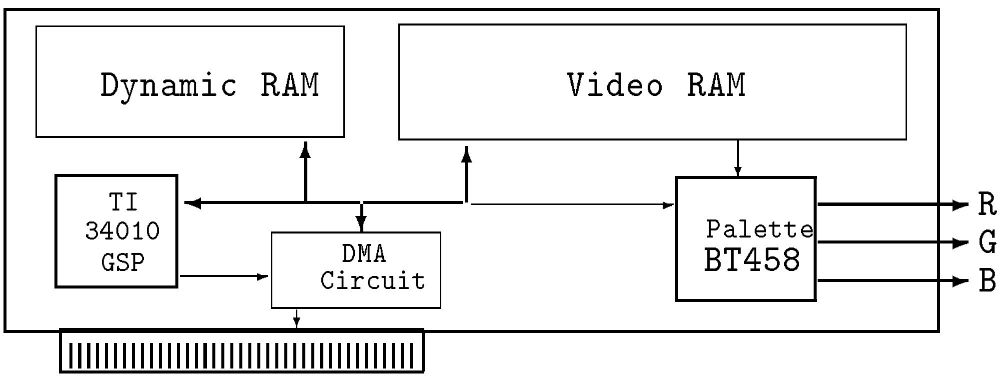

Figure 1: Block Diagram of the A2410 High Resolution Color Graphics Card

The Amiga as Host

The Amiga 2000 serves as the host for the High Resolution Graphics Card (A2410) and is used to manipulate and program the on-board TMS34010 registers, down-load new code and data into the frame buffer or into the GSP's local memory, and send messages between applications running on the host and the graphics manager running on the board.

The A2410 Graphics Card plugs into any of the Amiga Zorro II 100-pin expansion slots. As the Amiga is booting, it automatically assigns an address to the card via the standard amiga *auto-configuration* protocol.

High Resolution Color Graphics Card

This description of the hardware is provided to give an understanding of the graphics card architecture. Most programmers will not need this information because the device level software interface to the graphics card provides a higher level abstract interface to the A2410 functionality.

The six main functional blocks of the board are depicted in Figure 1 and include:

1. the Graphics System Processor (34010)
2. Video Memory (frame buffer) for images
3. Dynamic Memory (for programs and local data)
4. the Brooktree Palette chip
5. Control Register
6. DMA circuit

The heart of the graphics system is a highly integrated CPU, the TMS34010 *graphics system processor* (GSP), with an instruction set tailored for graphics applications. The GSP is responsible for communicating with the host, executing graphics instructions, refreshing memory, and refreshing the screen. The TMS34010 is a powerful CPU which combines the features of a general-purpose processor and a graphics controller. The TMS34010 instruction set includes a full complement of general purpose instructions, as well as graphics functions, from which you can construct efficient high-level functions. The instructions support arithmetic and boolean operations, data moves, conditional jumps, subroutine calls and returns.

There is video memory for image display that supports 1024 by 1024 eight-bit pixels and additional memory for two overlay bit-planes. In addition to this image frame buffer memory there is a dynamic memory block for storing program code and data.

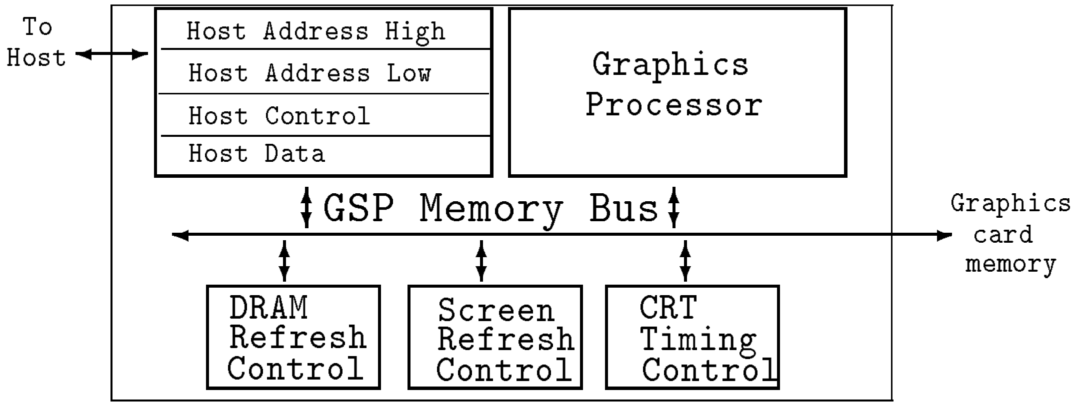

Figure 2: Graphics System Processor

The Graphics System Processor

All communication between the board and the Amiga is done through the GSP registers. The functional blocks of the GSP are show in Figure 2. The **host interface** consists of the *decode* and *auto-configuration* logic responsible for causing the graphics GSP to do one of several functions (depending on the address issued to it). These addresses correspond to the GSP's four host registers:

HSTADRL Host address low

HSTADRH Host address high

HSTDATA Host data

HSTCTL host control

The HSTADRL and HSTADRH registers can be loaded with the low and high 16-bit words, respectively, of a 32-bit address pointer in GSP's local memory. They can be used by the Amiga to access or load data and programs into the dynamic memory to be executed by the GSP. In addition, the image to be displayed is moved from system memory into the frame buffer by loading these two registers with the address of a starting location within video memory. The HSTDATA register will contain the 16-bit data to be read or written to GSP's memory. HSTCTL is a control register containing bits for *interrupt requests* and *status codes* between the host and GSP.

The address pointers should be loaded into the GSP's HSTADRL and HSTADRH registers before doing one or more accesses of local memory using the HSTDATA register. To ensure pointing to a word boundary, the GSP rounds the four LSB's of HSTADRL to 0's. Also, in order to access a block of data without the overhead of incrementing each time, the GSP can be put in an *auto-increment mode*. This is done by setting appropriate bits in the control register (INCR,INRW). By loading the address pointer, an update of HSTDATA

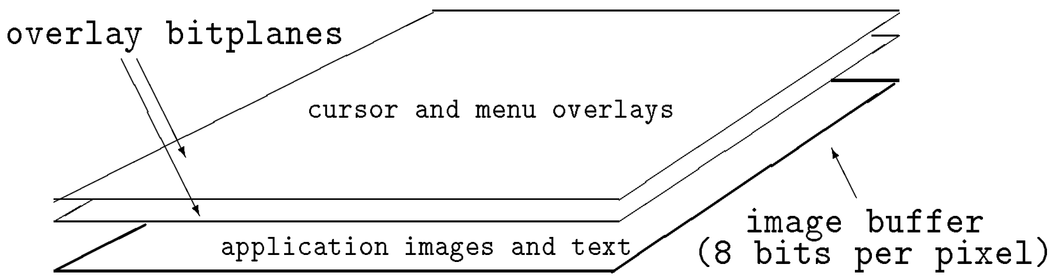

Figure 3: Video Memory Configuration

is automatically triggered. Each subsequent host access of HSTDATA will cause HSTADL and HSTADRH to be automatically incremented (if INCR or INRW are set) to point to the next word location in memory. Note that there is no hardware stop to prevent the simultaneous access of the host registers by the GSP and Amiga. Software written must avoid this situation to prevent invalid data in the registers.

Video Memory (Frame Buffer)

In graphics systems today, there are several major methods of representing frame buffer data and latching it through the digital to analog converters that drive RGB monitors. One method is known as *bit-plane organization*, and has separate planes of memory for each bit of every pixel in the video memory. This method is used in the native amiga graphics environment.

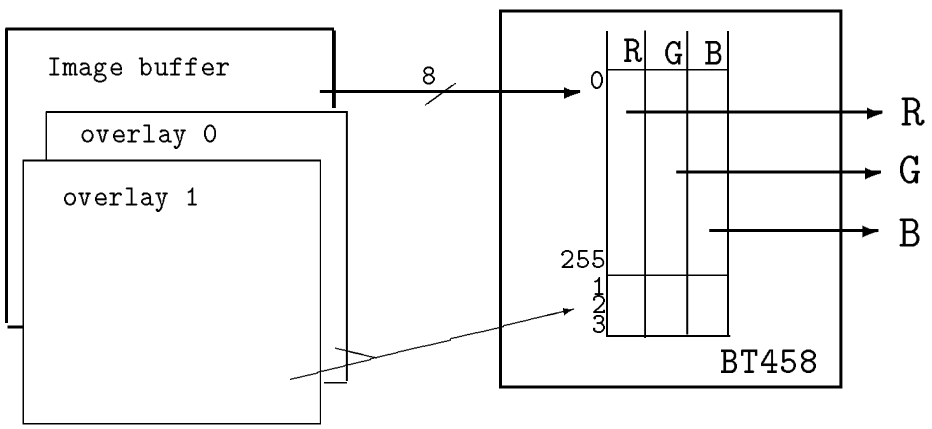

Figure 4: Color assignment through the BT458 palette chip

On the GSP board the *chunky mode* method is used. The *pixel* data is arranged contiguously in memory as consecutive eight-bit values. Additionally, there are two bit-planes of data that are used as overlay planes. The arrangement of the frame buffer and overlay planes can be seen in Figure 3.

Each eight-bit pixel in the image frame buffer is used as a pointer into a 256-element look-up table. In the look-up table the pixel value is assigned a 24-bit RGB color value. A Brooktree palette chip (BT458) provides this look-up table on the A2410. The BT458 also supports a separate look-up table that is used by the overlay bitplanes. The arrangement is show in Figure 4.

Control Register

The A2410 Graphics Card has a number of software features that utilize a special programmable register to allow the software to configure the system to match its current needs. The first of these allow the software to enable the DMA circuitry (which defaults to an idle state following a hardware reset). Once this circuitry is enabled, the hardware will automatically initiate the proper control signal protocol to transfer data between the A2410 and the Amiga. The circuit can transfer multiple words per bus access. The second mode bit enables the board to perform a byte-swap operation on all data that is being transferred to or from the High Resolution Graphics Card. This allows high speed data transfer to continue without the need to realign the most and least significant bytes while moving data between the host processor and graphics processor. The next function takes advantage of the Flash Write Enable signal (FWE) available in many commercial Video RAMs (VRAMS). By allowing the utilization of this feature, the screen can be cleared in a small fraction of the time that would be required by the conventional method of sequentially writing to the frame buffer memory. The remaining functionality of the programmable register concerns the actual display functionality of the A2410 High Resolution Graphics Card. The A2410 has two video clocks to accommodate a wide variety of resolutions. One clock is used for high resolutions and a mode bit in the control register can activate an alternate clock to handle NTSC or PAL interlaced video scan rates. Finally, the register is responsible for allowing the user to set the mode of sync signal necessary for board compatability with whatever monitor/cable combination is desired. This would permit the software control of selecting the type of sync required; Composite or seperate Horizontal/Vertical, as well as the output line format; Sync on Green or seperate lines.

Texas Instruments Graphics Architecture

The *Texas Instruments Graphics Architecture* TIGA is a software interface between an application program running on a host computer and a graphics board based on the TMS340x0 family of graphics processors. An implementation of TIGA as an Amiga device acts as the low level programming interface for the high resolution graphics card. In this version of TIGA the host is an Amiga, and the graphics board is the high resolution graphics card, but an application written using TIGA will be portable to any TIGA- compatible system. Target applications include computer-aided design, desktop publishing, imaging, and presentation graphics, which naturally benefit from the high speed, high resolution, portability,

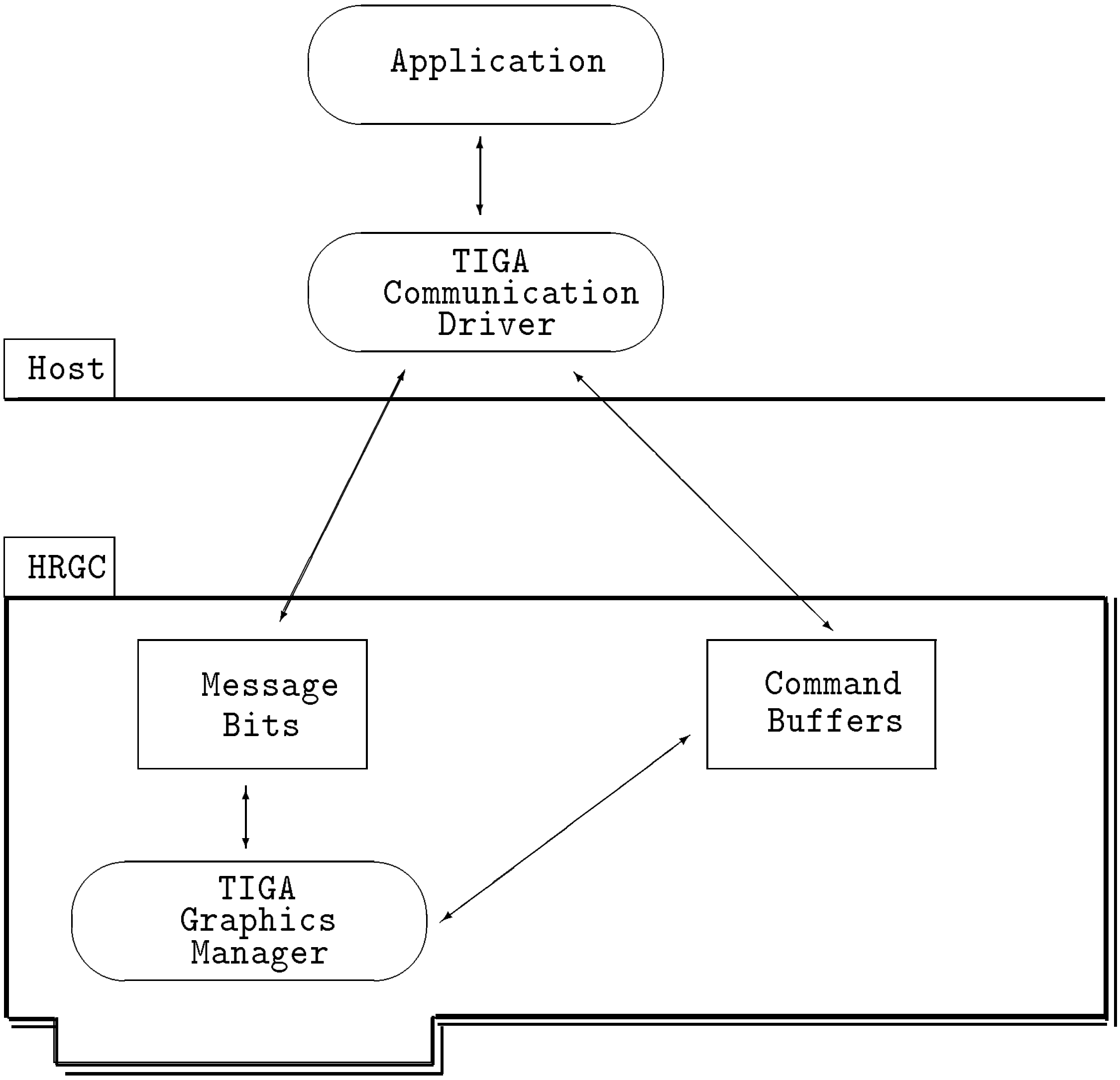

Figure 5: TIGA Architecture

and flexibility provided by TIGA.

As shown in Figure 5, the two main components of TIGA are the *communications driver* which runs on the Amiga, and the *graphics manager* which runs on the graphics processor. The TIGA interface is a protocol which utilizes a circular queue on the graphics processor to buffer up commands as they are received from the host. The Amiga implementation of TIGA is done as an Exec level device. Thus in most situations the application program can resume processing as soon as the graphics command is sent, without waiting for the command to be executed.

The communications driver handles the handshaking necessary to initiate communication with the graphics processor, to send and receive commands and data between the GSP and the host. Synchronization and buffer allocation are performed jointly by the communications driver and by the graphics manager, and are therefore transparent to the application program.

The graphics manager consists of the command processor which communicates directly with the communications driver, a C-Packet handler, and a suite of graphics processor functions. These functions can be partitioned either by calling mode, or by functionality.

The graphics processor functions are written in either TMS34010 assembly language, in which case they are called via *direct mode* calls; or in C, in which case they are called via *C-packet* calls. Each time a command is received from the host, the command processor determines which type of call is appropriate. If the current command is of the direct mode type, the corresponding function is called directly from the command processor. If the current command is of the C-Packet type, the C-Packet handler is called to stack the arguments according to C convention, and then the corresponding C function is called. This is all handled by the graphics manager's command processor on the high resolution graphics card.

The graphics processor functions are functionally partitioned into *Core Primitives* and *Extended Primitives.* The Core primitives consist of graphics system initialization functions such as init-palet, graphics attribute control functions such as those to set pixel processing operation and to enable or disable transparency, text functions, cursor functions, memory management functions, and functions to send raw data between the host and the graphics processor. The Extended primitives consist mainly of graphics output functions, such as draw-line, draw-rect, seed-fill, etc. So, if an application were designed to use the board for image processing, for example, it would be possible to load in the core primitives and substitute application-specific routines for the graphics output functions.

The following tables list the functions accessible via the Exec device interface to TIGA. More information about programming the TIGA device is supplied with the documents and examples that accompany the developer releases of the high resolution graphics card.

Function	*Description*	*Type*
Graphics System Initialization		
cd_is_alive	Return if TIGACD is running	Core
function_implemented	Return if a function is implemented	Core
get_config	Return board configuration	Core
get_modeinfo	Return board configuration	Core
get_videomode	Return current emulation mode	Core
gsp_execute	Execute a COFF program	Core
install_primitives	Install extended drawing primitives	Core
install_usererror	Install user error	Core
loadcoff	Load a COFF program	Core
set_config	Set graphics config	Core
set_timeout	Set timeout timing value	Core
set_videomode	Set emulation mode	Core
synchronize	Make host wait for GSP to idle	Core

Function	*Description*	*Type*
Clear Functions		
clear_frame_buffer	Clear entire frame buffer	Core
clear_page	Clear current drawing page	Core
clear_screen	Clear screen	Core

Function	*Description*	*Type*
Graphics Attribute Control		
cpw	Compare point to window	Core
get_colors	Returns fore- and background colors	Core
get_env	Returns current environment structure	Core
get_pmask	Returns color plane mask	Core
get_ppop	Returns pixel processing operation	Core
get_transp	Returns transparency mode	Core
get_windowing	Inquire windowing mode	Core
set_bcolor	Set background color	Core
set_clip_rect	Set clipping rectangle	Core
set_colors	Set foreground and background colors	Core
set_draw_origin	Set drawing origin	Ext
set_fcolor	Sets foreground color	Core
set_pattn_addr	Sets address of current pattern	Ext
set_pensize	Sets current pensize	Ext
set_pmask	Sets color plane mask	Core
set_ppop	Sets pixel processing operation	Core
set_transp	Set transparency mode	Core
set_windowing	Sets windowing mode	Core
transp_off	Disables pixel transparency	Core
transp_on	Enables pixel transparency	Core

Function	*Description*	*Type*
Palette Functions		
get_nearest_color	Return nearest color in palette	Core
get_palet	Return an entire palette	Core
get_palete_entry	Return a palette entry	Core
init_palet	Default palette	Core
set_palet	Set an entire palette	Core
set_palet_entry	Set a palette entry	Core

Function	*Description*	*Type*
Cursor Functions		
get_curs_state	Return cursor current state	Core
get_curs_xy	Return cursor position	Core
set_curs_shape	Set cursor shape	Core
set_curs_state	Make cursor visible/invisible	Core
set_curs_xy	Set current cursor position	Core

Function	*Description*	*Type*
Communication Functions		
cop2gsp	Copy coprocessor to GSP memory	Core
field_extract	Extract data from GSP memory	Core
field_insert	Insert data into GSP memory	Core
gsp2cop	Copy GSP memory to coprocessor	Core
gsp2host	Copy from GSP into host memory	Core
gsp2hostxy	Copy rectangular area from GSP to host	Core
host2gsp	Copy from host into GSP memory	Core
host2gspxy	Copy rectangular area from host to GSP	Core

Function	*Description*	*Type*
Extensibility Functions		
create_aim	Create absolute load module	Core
create_esym	Create external symbol table file	Core
flush_esym	Flush external symbol table file	Core
flush_extended	Flush all user extensions	Core
get_isr_priorities	Return interrupt service routine priorities	Core
install_alm	Install absolute load module	Core
install_primitives	Install extended drawing primitives	Core
install_rim	Install relocatable load module	Core
set_interrupt	Set an interrupt handler	Core

Function	*Description*	*Type*
Graphics Output		
draw_line	Draw line	Ext
draw_oval	Draw ellipse outline	Ext
draw_ovalarc	Draw ellipse arc	Ext
draw_piearc	Draw ellipse pie slice	Ext
draw_point	Draw single pixel	Ext
draw_polyline	Draw list of lines	Ext
draw_rect	Draw rectangle outline	Ext
fill_convex	Draw solid convex polygon	Ext
fill_oval	Draw solid ellipse	Ext
fill_piearc	Draw solid ellipse pie slice	Ext
fill_polygon	Draw solid polygon	Ext
fill_rect	Draw solid rectangle	Ext
frame_oval	Draw oval border	Ext
frame_rect	Draw rectangular border	Ext
patnfill_convex	Draw patterned convex polygon	Ext
patnfill_oval	Draw patterned ellipse	Ext
patnfill_piearc	Draw patterned pie slice	Ext
patnfill_polygon	Draw patterned polygon	Ext
patnfill_rect	Draw patterned rectangular	Ext
patnframe_oval	Draw patterned oval border	Ext
patnframe_rect	Draw patterned rectangular border	Ext
patnpen_line	Draw line with pattern and pen	Ext
patnpen_ovalarc	Draw oval arc with pattern and pen	Ext
patnpen_piearc	Draw pie slice with pattern and pen	Ext
patnpen_point	Draw pixel with pattern and pen	Ext
patnpen_polyline	Draw lines with pattern and pen	Ext
pen_line	Draw line with pen	Ext
pen_ovalarc	Draw an oval arc with pen	Ext
pen_piearc	Draw pie slice with pen	Ext
pen_point	Draw point with pen	Ext
pen_polyline	Draw lines with pen	Ext
seed_fill	Fill region with color	Ext
styled_line	Draw styled line	Ext

Function	*Description*	*Type*
Poly Drawing Functions		
draw_polyline	Draw polyline	Ext
fill_convex	Fill convex polygon	Ext
fill_polygon	Fill polygon	Ext
patnfill_convex	Pattern fill convex	Ext
patnfill_polygon	Pattern fill polygon	Ext
patnpen_polyline	Pattern pen polyline	Ext
pen_polyline	Pen polyline	Ext

Function	*Description*	*Type*
Workspace Functions		
fill_piearc	Fill pie arc	Ext
fill_polygon	Fill polygon	Ext
get_wksp	Return offscreen workspace	Core
patnfill_piearc	Pattern fill pie arc	Ext
patnfill_polygon	Pattern fill polygon	Ext
set_wksp	Set a temporary workspace	Core

Function	*Description*	*Type*
Pixel Array Functions		
bitbit	Bitbit source array to destination	Ext
set_dstbm	Set destination bitmap	Ext
set_srcbm	Set source bitmap	Ext
swap_bm	Swap source and destination bitmaps	Ext
zoom_rect	Zoom source rectangle	Ext

Function	*Description*	*Type*
Text Functions		
delete_font	Remove a font from the font table	Ext
get_fontinfo	Return font physical information	Core
get_textattr	Return text rendering attributes	Ext
init_text	Initialize text drawing environment	Core
install_font	Install font into font table	Ext
select_font	Select an installed font for use	Ext
set_textattr	Set text rendering attributes	Ext
text_out	Render an ASCII string	Core
text_width	Return the width of an ASCII string	Ext

Function	*Description*	*Type*
Graphics Utility		
get_pixel	Read contents of a pixel	Ext
lmo	Return left-most-one bit number	Core
page_busy	Return status of page flipping	Core
page_flip	Set display and drawing pages	Core
peek_breg	Read from a B-file register	Core
poke-breg	Write to a B-file register	Core
rmo	Return right-most-one bit number	Core
wait_scan	Wait for a designated scan-line	Core

Function	*Description*	*Type*
Memory Management		
get_max_freespace	Return largest free block	Core
get_offscreen_memory	Return offscreen memory blocks	Core
gsp2gsp	Copy from GSP Memory to GSP Memory	Core
gsp_calloc	Allocate and clear GSP memory	Core
gsp_free	Deallocate GSP memory	Core
gsp_malloc	Allocate GSP memory	Core
gsp_minit	Reinitialize GSP memory heap pool	Core
gsp_realloc	Resize allocated block of memory	Core

References

[TI 88] Texas Instruments, *TMS34010 User's Guide,* Texas Instruments Incorporated, Houston TX, 1988

[TI 89] Texas Instruments, *TIGA-340 Interface User's Guide,* Texas Instruments Incorporated, Houston TX, 1989