THE PS 390 GRAPHICS SYSTEM
Preliminary Version 1.2

Evans & Sutherland

May 6, 1987

* * * NOTICE * * *
EVANS & SUTHERLAND assumes no responsibility for errors or inaccuracies in this document. The contents of this document are proprietary and are not to be reproduced or copied, in whole or in part, without the prior written permission of EVANS & SUTHERLAND. PS 300, PS 340, PS 350 and PS 390 are trademarks of EVANS & SUTHERLAND COMPUTER CORPORATION.

PS 390
HARDWARE REFERENCE MANUAL

SECOND EDITION

EVANS & SUTHERLAND
INTERACTIVE SYSTEMS DIVISION
580 ARAPEEN DRIVE
SALT LAKE CITY, UTAH 84108
PHONE # (801) 582-5847

UPDATE AND REVISION NOTIFICATION

This manual is technically accurate and current to the best of our knowledge. However, typographical errors and engineering change orders may necessitate revisions and updates from time to time. For this purpose, a REVISION LOGSHEET is provided with this manual. Please log the said changes in this manual as they become available.

EVANS & SUTHERLAND appreciates your comments and/or any noted corrections, inaccuracies or important omissions in this book. Please address your comments or inquiries to:

EVANS & SUTHERLAND
INTERACTIVE SYSTEMS DIVISION, PUBLICATIONS
580 ARAPEEN DRIVE
SALT LAKE CITY, UTAH 84108
PHONE # (801) 582-5847

REVISION LOGSHEET

UPDATE NUMBER	SECTION NUMBER	DATE	PAGES EFFECTED	UPDATE PLACED IN MANUAL	INITIALS

Contents

List of Figures

List of Tables

Part I

Internal Engineering

Chapter 1

PS 390 Functional Description

1.1 Overview

The PS 390 Graphics System provides dynamic *anti-aliased* lines on a Raster display using Shadowfax VLSI technology. The PS 390 Graphics System replaces the Refresh Buffer card, Color card, and the Line Generator card of a PS 350 system with two cards which comprise the Raster Backend portion of the PS 390 Graphics System. The Calligraphic Display from the PS 350 is replaced with a 1024 $\times$ 864 RGB raster display. (FIMI 2054C)

The PS 390 Graphics System is built on PS 300 sized cards. The main components of the PS 390 Graphics System are as follows:

- The Joint Control Processor (JCP)
- The Arithmetic Control Processor (ACP)
- The Pipeline Subsystem (PLS)
- Raster Backend Left Card (LC)
  - The Input FIFO.
  - The Master Bitslice Processor.
  - The Endpoint Pipeline.
  - The Pixel Read Machine.
  - 8 Pixel Processors.
  - Frame Buffer.
- Raster Backend Right Card (RC)

- 8 Pixel Processors.
- Frame Buffer.
- The Video Output System.

The main components of the PS 390 Graphics System are grouped together as shown in Figure 1.1.

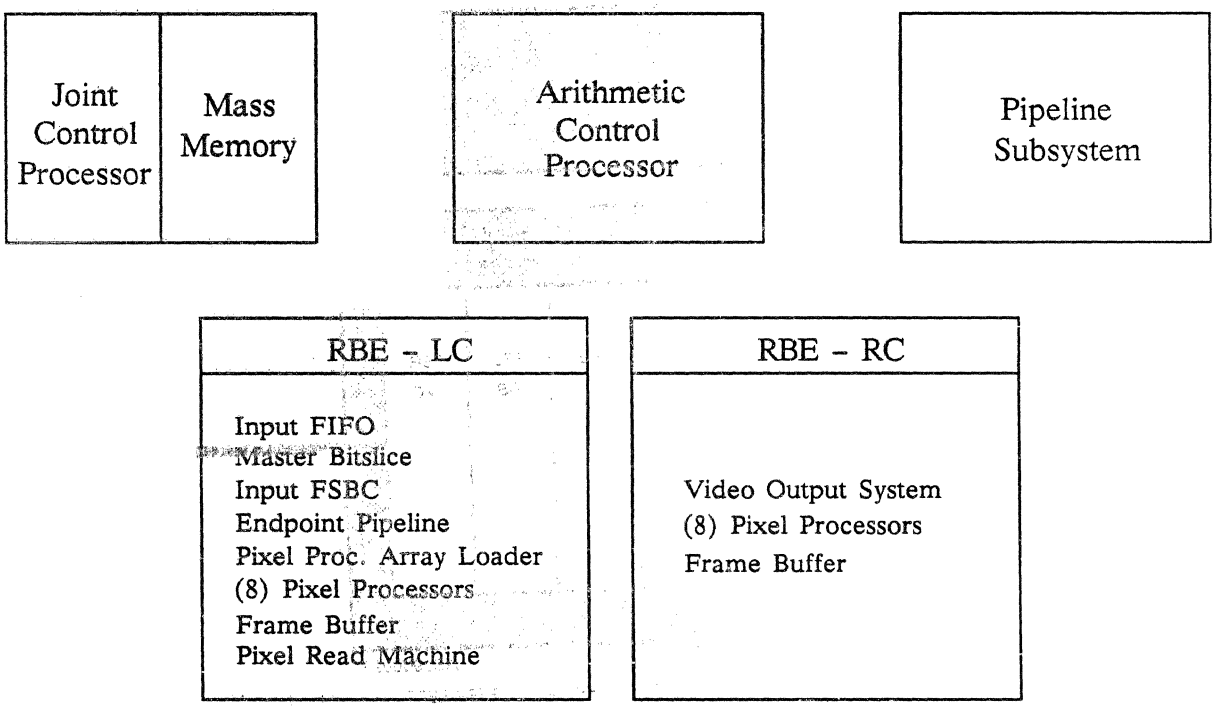


Figure 1.1: PS 390 Card Components

The PS 390 Raster Backend receives screen space end points and commands from the Pipeline Subsystem.

Endpoints are processed in several subcomponents of the PS 390 to produce the pixel information that is stored in the PS 390 Graphics System Frame Buffer.

Commands from the Pipeline Subsystem are used to configure hardware components of the Raster Backend and to instruct the Bitslice to retrieve pixel information residing in the Frame Buffer.

1.2 Data Flow Through the PS 390

Figure 1.2 shows a block diagram for the PS 390. The diagram illustrates the basic data flow through the system.

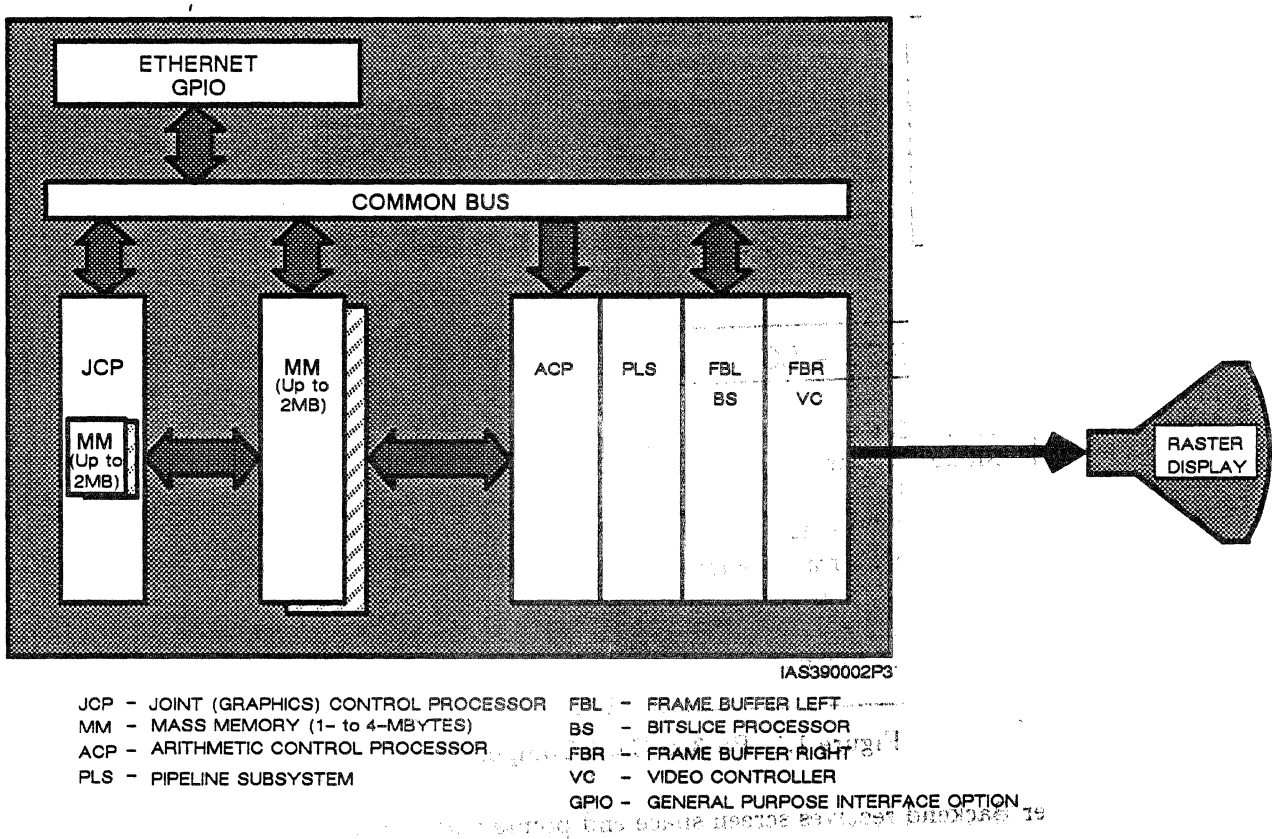


Figure 1.2: Data Flow through the PS 390

Data flow is bidirectional between the host and the PS 390's Mass Memory. The downloaded data may be affected locally (by the PS 390) or through interaction with the host computer. This section describes the PS 390 data flow as commands and data are input to the system and are processed for display.

1.2.1 Host Level Data Structure

The host sends the PS 390 ASCII or binary commands from user application programs or files that reside on the host. These commands are transferred over a communications interface to the PS 390. The standard communications interface is the RS-232C line. High-speed communications interfaces include the VAX UNIBUS Parallel Interface, the IBM 3278 Interface, the IBM 5080 Interface, and ETHERNET/DECNET.

Once ASCII commands are transferred to the PS 390, the Joint Control Processor (JCP) firmware parses and interprets the commands. ASCII commands pass from the interface to the Parser and the Command Interpreter. Binary commands sent from the host pass directly to the Command Interpreter for conversion to locally used formats.

1.2.2 Data Flow From Interactive Devices

Data are sent from the interactive devices to the PS 390 in serial RS-232C ASCII format.

The PS 390 offers a "function network" programming language. One of the purposes of the Function Networks is to process data from the interactive devices. Function networks specify which interactive devices provide rotation, scale and translate data to the display structures. Function networks also allow the interactive devices to communicate with user-specified programs in the host.

Using function networks, the user can specify smoothing, clamping, and other filtering operations on the raw data from the interactive devices before sending the data to the host or using them to modify display structures. There is a separate function type for each of the interactive devices. Data are routed from the interactive devices through these functions, through other (if any) user-specified functions and finally to the display structures or to the host. From this point, interactive device data flow in the same way as host data.

Chapter 2

Raster Backend Hardware Descriptions

2.1 The Input FIFO

The Input FIFO on the Raster Backend Left Card (LC) receives screen space endpoints, instructions and commands from the PS 390 Pipeline Subsystem via the existing Refresh Buffer interface. The FIFO can store up to 340 endpoints. The interface to the Pipeline Subsystem is designed so that when the FIFO is full the PLS stops the Arithmetic Control Processor until more data can be loaded into the Raster Backend Input FIFO. The main purpose for the Input FIFO is to allow the Arithmetic Control Processor to continue processing even if the Raster Backend is not ready to receive the next endpoint/command.

The Input FIFO hardware includes:

- A State Machine
- A 1K deep $\times$ 16-bit FIFO
- FIFO Buffer Register.

The Input FIFO State Machine consists of two parts. The FIFO Input State Machine which transfers data from the PLS into the FIFO, and the FIFO Output State Machine which transfers data from the FIFO into the FIFO buffer register.

The FIFO Input State Machine monitors the PLSREADY signal that is provided from the PLS. When the PLS signal is asserted, the state machine transfers three 16 bit words of data from the PLS to the FIFO. The state machine then asserts the RBDONE signal, which signals

to the PLS that the data transfer is complete. Then, the state machine returns to the idle state where it remains until the PLSREADY signal is again asserted by the PLS.

The Output FIFO State Machine monitors the FIFO empty bit. When the FIFO empty bit is negated, signifying that there is data ready in the FIFO, the Output State Machine transfers one 16 bit word of data to the FIFO buffer register and asserts the FIFOREADY signal for the Bitslice. After the Bitslice reads the FIFO Buffer register, the state machine transfers the next 16 bit word to the buffer register, provided that there is more data available in the FIFO. If no more data is available in the FIFO the Output State Machine returns to its idle state where it continues to wait for the FifoEmpty signal to be negated.

2.2 The Master Bitslice Processor

The PS 390 Graphics System Master Bitslice Processor obtains endpoints and commands from the Input FIFO. Endpoints are formatted into a packet of data and then sent to the PS 390 EndPoint Pipeline. The Bitslice Processor is located on the Raster Backend Left Card.

When instructions are passed through to the Master Bitslice processor, it decodes and executes them.

The Master Bitslice can interrupt the Graphics Control Processor on the JCP via the Common Bus to report requested information. The Master Bitslice can have mastership of the Common Bus.

The Master Bitslice has an interface to the Video Output System over the Y-Bus via a 74AS652 transceiver. This interface is mainly used to configure hardware circuitry for operation, to retrieve latched screen XY on lightpen hits, and for diagnostic purposes.

The Master Bitslice also collaborates with the Pixel Processor Array to operate on the PS 390 Graphics System Frame Buffer. During normal endpoint processing, the Master Bitslice sends data down the EndPoint Pipeline to the Pixel Processor Array. The Master Bitslice can also instruct the Pixel Processor Array to perform direct read and write operations to and from the Frame Buffer via the Pixel Read Machine, but must obtain permission from the PPALOADER.

The Master Bitslice Processor has the following hardware components:

- Bus Interfaces
  - Y-Bus
  - D-Bus
  - Immediate Bus

  - Branch Bus
- Bus Decoders
- Bitslice Processor. 16-Bit AMD 29117 Microprocessor
- Sequencer. 12-Bit AMD 2910A or equivalent
- Execution Register
- Writable Control Store. (4K $\times$ 80 bits)
- Scratch Ram. (2K $\times$ 16 bits)
- Multiplier, 16 $\times$ 16, parallel. WTL 2517B or equivalent.
- Writable Immediate Field Register (Wimmed)
- Vector Ram
- Lookup Tables. (64K $\times$ 16)
- Maintenance Register.
- Common Bus Interface. Mastery of Common Bus.
- Interface to Video Controller circuitry.

Figure 2.1 shows a block diagram of the Master Bitslice.

2.2.1 Bus Interfaces

The four Busses which interface with the Master Bitslice are:

- Y-Bus
- D-Bus
- Immediate Bus
- Branch Bus

2.2.1.1 The Y-Bus

The primary function of the Y-Bus is to pass data from the 29117 to various destinations in the Bitslice. Secondary sources of data to the Y-Bus include the D-Bus, Scan Line Buffer Interface and the Video Control Interface.

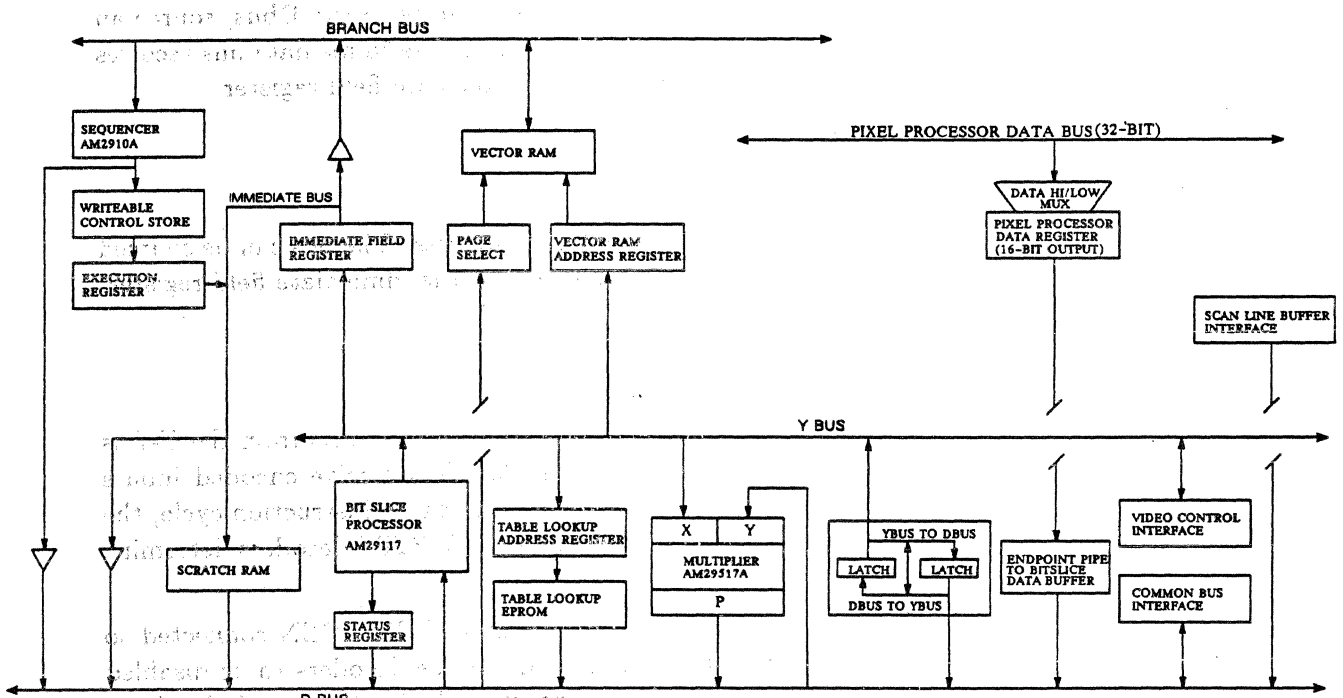


Figure 2.1: Block Diagram of the PS 390 Master Bitslice

2.2.1.2 The D-Bus

The primary function of the D-Bus is to pass data from various sources in the Bitslice to the 29117, the Y input of the multiplier and the Y-Bus via the bus to bus transceivers.

2.2.1.3 The Immediate Bus

The immediate bus is a multi-purpose bus. It can source data values to the Dbus, source an address to the Branch Bus and source the Scratch RAM address. The Immediate bus receives values from the execution registers immediate field and the immediate field register.

2.2.1.4 The Branch Bus

The Branch Bus supplies the microsequencer with an external address. The bus can be sourced from the vector RAM, the execution registers immediate field or the immediate field register.

2.2.1.5 Bus Decoders

All the components surrounding the Bitslice that send data to or receive data from the Y-Bus and D-Bus are controlled by the Bus Decoders. The Bus Decoders receive encoded inputs from one of four bus control fields in the Execution Register. During an instruction cycle, the READ decoders select which device puts data on the bus, and the WRITE decoders determine which device receives data from the bus.

The Y-Bus and D-Bus destination decoders have *STROBE and *WRDECEN connected to their enables. *WRDECEN from the maintenance register allows the decoders to be disabled while testing and loading the WCS to prevent the data's execution during these periods. Some decoder outputs are used to clock registers. A glitch on those lines will cause the registers to load. The *STROBE disables the decoders for the first 25ns of the instruction cycle. This gives the decoder time to stabilize before its outputs are enabled to prevent the outputs from glitching.

2.2.1.5.1 Bus-to-Bus Communication Communication between the Y-Bus and D-Bus is accomplished through a 74AS652 Bus Transceiver. The transceiver's mode of operation is selected from the microcode via the D-Bus Y-Bus decoders. The transceivers are controlled from the microcode so that the data can be latched internally and used in a subsequent state or they can be put in a transparent mode and the data passed through the transceiver in one state. For some data paths, timing will not allow the transceivers to operate in transparent mode.

For details on bus to bus timing constraints, refer to the PS 390 Raster Backend Microcode Manual.

2.2.2 The Bitslice 16-Bit Processor

The Master Bitslice Arithmetic Logic Unit (ALU) is an AMD 29117 or equivalent microprocessor. The AM29117 has dual data ports, ability to do register-to-register arithmetic and logic functions, and 16-bit barrel shift capability. Refer to the PS 390 Raster Backend Microcode Manual for details on programming the AM29117. Refer to the AM29117 manual for modes of operation and timing specifications.

The Bitslice has been connected to maximize control via the microcode. Input data is supplied from the Raster Backend D-bus. The output data is sent to the Raster Backend Y-bus. The Data Latch Enable bit controlling the reading of data from the D-bus to the 29117 internal latch is controller directly by the execution register. The (OEY) output enable (which controls the sourcing of data to the Y-bus) is controlled through the Y-bus source decoders.

2.2.2.1 29117 Bitslice ALU Pin Description

The following is a description of the pin assignments for the 29117 ALU chip:

- $D_0 \rightarrow D_{15}$ Bidirectional Data (Input)
  - Data Input Lines, $D_0 \rightarrow D_{15}$, are used as external data inputs which allow data to be directly loaded into the 16-bit data latch.
- $Y_0 \rightarrow Y_{15}$ General Output (Output)
  - Data Output lines. When $\overline{OE_Y}$ is HIGH, the 16-bit Y outputs are disabled (high impedance); having $\overline{OE_Y}$ LOW allows the ALU data to be output on $Y_0 \rightarrow Y_{15}$.
- DLE Data Latch Enable (Input)
  - When $\overline{OE_Y}$ is HIGH, the 16-bit data latch is transparent and is latched when DLE is LOW.
- $\overline{OE_Y}$ $\overline{OutputEnable}$ (Input)
  - When $\overline{OE_Y}$ is HIGH, the 16-bit Y outputs are disabled (high impedance); when $\overline{OE_Y}$ is LOW, the 16-bit Y outputs are enabled (HIGH or LOW).
- $I_0 \rightarrow I_{15}$ Instruction Inputs (Input)

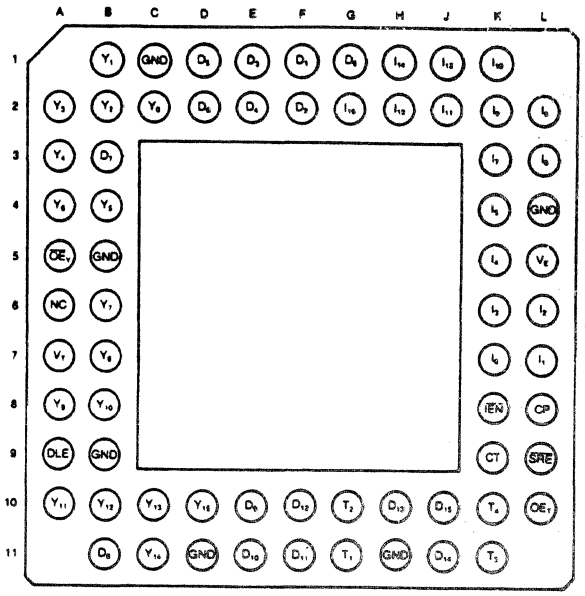


Figure 2.2: 29117 Bitslice ALU Pin Assignments

- Sixteen Instruction Inputs, used to select the operation to be performed in the 29117. Also used as data inputs while performing immediate instructions.

- $\overline{IEN}$ *InstructionEnable* (Input)
  - When $\overline{IEN}$ is LOW, data can be written into RAM when the clock is LOW. The Accumulator cam accept data during the LOW-to-HIGH transition of the clock. Having $\overline{IEN}$ LOW, the Status Register can be updated when $\overline{SRE}$ is LOW. With $\overline{IEN}$ high, the conditional test output, CT, is disabled as a function of the instruction inputs.
- $\overline{SRE}$ *StatusRegisterEnable* (Input)
  - When $\overline{SRE}$ and $\overline{IEN}$ are both LOW, the Status Register is updated at the end of all instructions with the exception of NO-OP, Save Status and Test Status. Having either $\overline{SRE}$ or $\overline{IEN}$ HIGH will inhibit the Status Register from changing.
- CP Clock Pulse (Input)

  - The clock input to the 29117. The RAM latch is transparent when the clock is HIGH. When the clock goes LOW, the RAM output is latched. Data is written into the RAM during the LOW period of the clock, provided $\overline{IEN}$ is LOW, and if the instruction being executed designates the RAM as the destination of the operation. The Accumulator and the Status Register will accept data on the LOW-to-HIGH transition of the clock if $\overline{IEN}$ is also LOW. The instruction latch becomes transparent when it exits an immediate instruction mode during a LOW-to-HIGH transition of the clock.

- $T_1 \rightarrow T_4$ Test I/O Pins (Input/Output)
  - Under the control of OE_T, the four lower status bits, Z, C, N and OVR become outputs on $T_1 \rightarrow T_4$, respectively, when OE_T goes HIGH. When OE_T is low, $T_1 \rightarrow T_4$ are used as inputs to generate the CT output.
- OE_T Output Enable (Output)
  - When OE_T is LOW, 4-bit T outputs are disabled (high impedance); when OE_T is HIGH, the 4-bit T outputs are enabled (HIGH or LOW).
- CT Conditional Test (Output)
  - The condition code multiplexer selects one of the twelve condition code signals and places it on the CT output. A HIGH on the CT output indicates a passed condition and a LOW indicates a failed condition.

2.2.2.2 Instructing the 29117 16-Bit Processor

The Enable bit ($\overline{IEN}$) is wired to a 50% duty cycle 100ns clock to include dual register instructions in the microcode instruction set. The execution register (refer to Section 2.2.4) has one set of registers specifying the 29117 source register address and a second set of registers specifying the 29117 destination register address. The register addresses run to the 29117 via a "registered" multiplexer that selects the source register address for the first half of the instruction and the destination register address for the second half.

NOTE: Because of timing constraints, the source register address does not have time to go through the execution register. The source register address runs from the WCS directly to the multiplexer. The destination registers are run to the multiplexer via the execution register to insure their integrity.

The Status Register Enable is directly wired to a bit in the execution register to allow the microcode to selectively update the 29117 internal status register. The (OET) Output Enable

is tied high to enable the status bits as outputs. The 4 status bits and the conditional test bit (CT) are wired to the external status register.

The external status register contains two banks of registers. The first bank containing the Endpoint Pipeline and pixel processor status bits is continually updated. The second bank, which receives the status bits from the 29117, can be selectively updated by the microcode.

Timing allows the 29117 internal status register and the external status register to be updated in the same state.

2.2.3 The Microsequencer

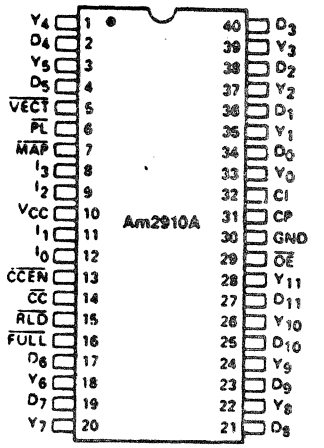


Figure 2.3: AM2910A Bitslice Microsequencer Pin Assignments

The microsequencer is an AM2910A or equivalent. The 12-bit microsequencer can select any of the 4K writable control store addresses. The microsequencer receives an instruction at the beginning of each cycle and using that instruction determines the address of the instruction to be used for the next cycle. The microsequencer instruction set contains commands such as:

- Continue to access the next sequential instruction
- Jump to a designated address
- Jump to a designated address on a specified condition being true.

The microsequencer receives its instructions directly from the execution register. Its outputs run to the writable control store address lines. The microsequencer is the only device capable of driving the writable control store address.

When executing a branch instruction, the address is provided from a source external to the microsequencer. The microsequencer receives the address on its input branch bus and routes the address to the WCS. In the PS 390 Raster Backend, a branch address can be sourced from the Y-Register's immediate field, the immediate field register, or the vector RAM. See the PS 390 Raster Backend Microcode Manual for details on selecting the various address sources.

A branch on condition is executed when the microsequencer's condition code enable bit is asserted LOW. When executing a branch on condition instructions, the status of the microsequencer's one bit condition code determines if the branch condition has failed or passed. This is achieved by connecting the Status Register to a multiplexer. The Multiplexer's output is connected to the microsequencer's condition code bit. The microsequencer must be capable of detecting and branching on the state of the various logical segments in the Raster Backend. The select lines of the multiplexer are controlled from the execution register allowing any of the status bits to be forwarded to the microsequencer condition code bit.

The microsequencer has the restriction that it can only branch on an OR condition. If the condition fails, the branch is not executed. To give the programmer the ability to branch on a failing condition as well as a passing condition, an exclusive OR gate is placed between the output of the condition select multiplexer and the microsequencer. One input of the exclusive OR gate is connected to the multiplexer and the other is connected to the invert condition bit of the execution register. The invert condition bit determines if the condition bit is passed from the multiplexer to the microsequencer unchanged by the exclusive OR gate or if the condition is inverted by the gate. Inverting the condition allows a failing condition to be seen as a passing condition by the microsequencer and thus provides a branch false capability.

2.2.3.1 2910 Bitslice Microsequencer Pin Description

The following is a description of the pin assignments for the AM2910A microsequencer chip:

- D_i Direct Input Bit i
  - Direct input to register/counter and multiplexer. D_0 is LSB.

- I_i Instruction Bit i
  - Selects one-of-sixteen instructions for the AM2910A.
- $\overline{CC}$ Condition Code
  - Used as test criterion. Pass test is a LOW on $\overline{CC}$.
- $\overline{CCEN}$
  - Whenever the signal is HIGH, $\overline{CC}$ is ignored and the part operates as though $\overline{CC}$ were true (LOW).
- CI Carry-In
  - Low order carry input to incrementer for microprogram counter.
- $\overline{RLD}$ Register Load
  - When LOW forces loading of register/counter regardless of instruction or condition.
- $\overline{OE}$ Output Enable
  - Three-state control of Y_i outputs.
- CP Clock Pulse
  - Triggers all internal state changes at LOW-to-HIGH edge.
- V_{CC} +5 Volts
- GND Ground
- Y_i Microprogram Address Bit i
  - Address to microprogram memory. Y_0 is LSB, Y_{11} is MSB.
- $\overline{FULL}$ Full
  - Indicates that five item are on the stack.
- $\overline{PL}$ Pipeline Address Enable
  - Can select #1 source (usually Pipeline Register) as direct input source.
- $\overline{MAP}$ Map Address Enable
  - Can select #2 source (usually Mapping PROM or PLA) as direct input source.
- $\overline{VECT}$ Vector Address Enable
  - Can select #3 source (for example, Interrupt Starting Address) as direct input source.

2.2.4 The Execution Register

Functionally, the execution register is a bank of registers that receive a microcode instruction from the Writable Control Store that is to be executed by the Bitslice. The execution register's purpose is to keep the microcode instruction stable for one complete clock cycle while the microsequencer and WCS prepare the next instruction for execution.

The execution register is a mixture of AM29818s and 74F374s. The 29818s are used as the execution register with the exception of the ALU instruction field. The propagation delay through the 29818s is too long for the ALU instruction field is too long for the ALU to meet the 29818's 100ns cycle time requirement. Therefore, 29818s are used to complement the 74F374s in loading the WCS ALU instruction field.

The output enable pin on the 29818s is grounded with the exception of the 29818s which are associated with the immediate field. This gives the microcode the capability to select between the immediate field register and the microcode's immediate field as sources to the immediate bus.

2.2.5 The Writable Control Store (4K Words)

The Writable Control Store (WCS) is a bank of 4K $\times$ 80, 25ns RAMs. Their address lines are driven by the microsequencer. The data lines run directly to the Execution Register. The write enable is connected to a bit in the Maintenance Register to give the JCP control over loading the Bitslice microcode. The write enable is only enabled while loading microcode to the writable control store. For normal run mode operations, the write enable is deselected (held high).

The chip select is controlled from two sources via an OR gate. When in maintenance mode, the RAMs can be de-selected from the *_WCSOE bit of the maintenance register. When loading or executing microcode, the *WCSOE bit is disabled and the clock *STROBE controls the chip select. *STROBE is used here, primarily, to simplify the process of loading the RAMs. Details on the timing of *STROBE are included in the clock section.

2.2.5.1 Loading the WCS

The WCS is loaded via the Execution Register's 29818 Shadow Register. Before loading the WCS, the microsequencer must be initialized to select the starting address. This is accomplished by loading a Jump Immediate instruction into the 29818 shadow pipe. The immediate field of the jump instruction should contain the first WCS address to be loaded; normally zero. Setting the appropriate control bits and executing a single step will load the execution register

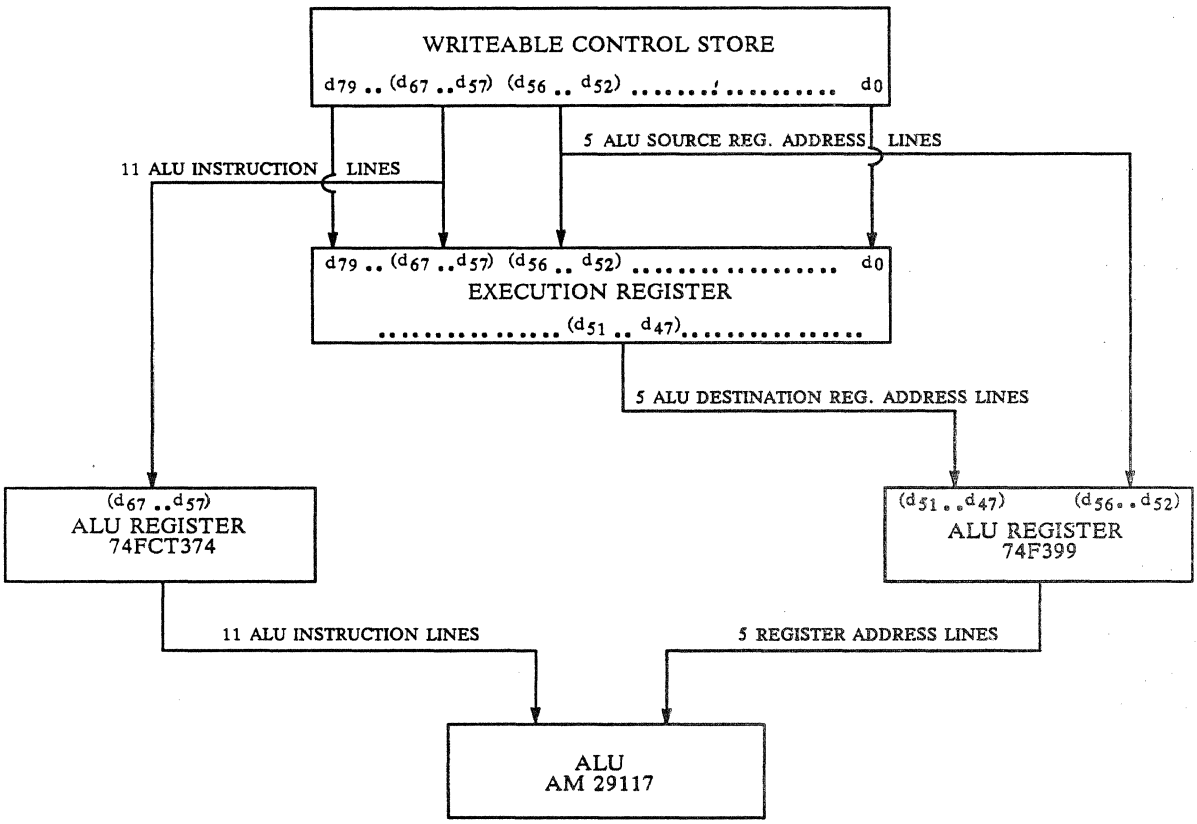


Figure 2.4: ALU Instruction Field Path

with the jump instruction in the shadow register. The microsequencer then accesses the first WCS location to be loaded. The first location must be loaded at this point by performing the following steps:

1. Load the shadow register with the data to be loaded in the WCS.
2. Drive the shadow register onto the WCS data lines by setting the appropriate mode bits and performing a D Clk.
3. Execute a P Clk to write the WCS.

The execution register should then be loaded with the default microword instruction (NOP). This will put the Bitslice in an idle state while the microcode is being loaded and allows the microsequencer to sequentially step through the WCS. This is accomplished by loading the shadow register with the default microword and executing another singlestep. The microsequencer is now accessing WCS location start + 1. The three steps outlined above can now be repeated to load WCS address start + 1 with its respective data. With a NOP instruction in the execution register, each time a P Clk is executed the microsequencer will increment its address by one. The three steps outlined above can be repeated until WCS is loaded.

The 29818s' Mode bit, Shift bit in and D Clock inputs are controlled from the maintenance register. This gives the JCP control over loading the WCS (This technique also prohibits the Bitslice from doing overlays in its microcode).

2.2.6 Scratch RAM

The Scratch RAM is 2K × 16-bits general purpose RAM (Random Access Memory). RAM addresses are supplied from the Immediate bus. Data lines are connected to the D-Bus. Timing constraints prohibit data from the ALU to be written directly to Scratch RAM. Data from the ALU must be transferred to the bus-to-bus transceivers in one state and from the transceivers to the Scratch RAM in a subsequent state. All other components can read/write data in the Scratch Ram in one state.

Data can be read from the Scratch RAM to the ALU D-latch in one state.

2.2.7 Multiplier

The multiplier is a Weitek 2517B or equivalent. The multiplier is wired to allow maximum control through the microcode. The execution register multiplier field has control lines for:

- X and Y Data Modes.

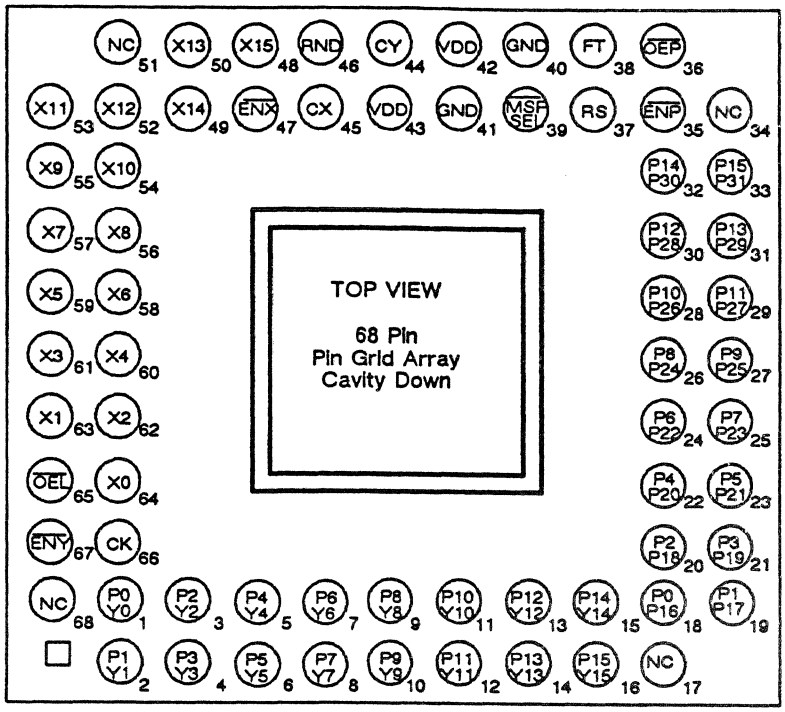


Figure 2.5: 2517B Bitslice Multiplier Pin Assignments

- Format Adjuster & Round Control.
- Feed-Through Mode Control.
- Product Enable.
- Most/Least Significant Product Select.

The multipliers X-data input is supplied through the Y-Bus. Loading of the X-data is controlled through the Y-Bus destination decoders. The multipliers Y-data is supplied through the D-Bus. Loading Y-data is controlled through the D-Bus destination decoders. The multipliers product is supplied to the D-Bus. Supplying the product to the D-Bus is controlled by the D-Bus source decoder.

Timing allows data to be written to the X-data input from any component on the Y-Bus or D-Bus in one state. Data being written to the Y-data input can be written from any component on the busses in one state with the exception of the ALU. The ALU must write the Y-data input via the bus transceivers temporary register. The product can be written to the ALU

two states after the data inputs are written. All other destinations can be written to from the ALU state one state after the operands are written.

2.2.7.1 2517B Bitslice Multiplier Signal Description

The following is a signal description for the Weitek 2517B Multiplier:

- X_I, X_I
  - 16 bits of data each, defined as unsigned magnitude or two's complement under control of C_X and C_Y. Y_I can also be used to output the LSP data.
- C_X, C_Y
  - X_I and Y_I complement control signals; a logic "1" on C_X or C_Y signifies a two's complement input (X_I or Y_I, respectively), while a logic "0" signifies an unsigned magnitude input.
- RND
  - A logic "1" on the RND line rounds the product to the most significant 16 bits by adding one to the most significant bit of LSP; RND is loaded by the rising edge of the logic "or" of CK_X and CK_Y.
- RS
  - When RS = 0, the MSP is left shifted one bit and the sign bit is duplicated in the MSB of the LSP. When RS is equal to one, P_{31} represents the product sign bit if the output is interpreted as two's complement data.
- FT
  - Feedthrough; makes output latch transparent when set high.
- P_I
  - 16 bits output data; Most Significant Product (P_{31} - P_{16}) and Least Significant Product ($P_{15} - P_0$) can be multiplexed onto the P port.
- CK_M, CK_L
  - MSP and LSP output register clocks.
- $\overline{OE}_P$, $\overline{OE}_L$
  - MSP and LSP three state output enables
- $\overline{MSPSEL}$
  - Select either MSP (low) or LSP (high) to be available at the product output port.

- $\overline{ENX}$
  - Register enable for X_{15} - X_0, C_X and RND.
- $\overline{ENY}$
  - Register enable for Y_{15} - Y_0, C_Y and RND.
- $\overline{ENP}$
  - Register enable for MSP and LSP.

Note: For more information concerning the signal descriptions for the WTL 2517 consult the manufacturer's chip specification document.

2.2.8 Wimmed Register

The Wimmed register is most often loaded with a branch address for the microsequencer. The Wimmed register is loaded from the Y-Bus and can source the immediate bus in any subsequent states. The microcode bit enabling the Wimmed register to the immediate bus is delayed by one state. If, for example, the microcode enables Wimmed register in states 2 and 3 of the microcode, its data will be on the immediate bus in states 3 and 4. This functionality provides the microcoder with a simple means of implementing jump tables.

2.2.9 Vector RAM

The Vector RAM is another source of branch addresses for the microsequencer. It can provide up to 2K of indirect addressing. The address bits are wired to allow paging through the RAM. The upper 3 bits are wired to a page select register, loaded from the D-Bus. The lower 8 bits are sourced from the entry select register, loaded from the Y-Bus.

Currently the PS 390 Raster Backend loads one page of the vector ram with addresses associated with the line texturing microcode and another page with the branch addresses of the command handling routines (which are received from the PLS).

The Vector RAM is loaded off disk during the boot process. If the microcode is modified altering the branch addresses. The Vector RAM file must be updated on the disk to correspond with the latest version of microcode.

As an example to the Vector RAMs operation consider the command handling routines. Data read from the Input FIFO, is loaded into the ALU D-latch and the Vector RAM address register. While the ALU performs a bit test to determine if the data is a command, the Vector RAM is providing a branch address on the assumption the data is a command. The

microsequencer performs a condition of branch on the result of the Bit Test using the address provided by the vector RAM as its branch address.

2.2.10 Lookup Tables

The function PROM is composed of 2, 64K × 8 UV EPROMS. Access time is 250ns. Timing requires three wait states after the Function PROM address register is written before the EPROM is sourced to the D-Bus.

The Function PROM provides a 64K × 16 bit look up table. Its contents currently consists of a sin (x) table, 1/X lookup table, Red, Green, and Blue color values and a vector list displayable during boot up.

2.2.11 Maintenance Register

The Maintenance register is loaded from the Mass Memory via the common bus. The common bus address lines along with *CBDEV, *CBWRT and *CBADS are run into the Maintenance Register Read/Write Decode PAL. The Decode PAL uses *CBADS to determine when the common bus address lines are stable and can be decoded. The address lines and *CBDEV determine if the maintenance register has been addressed (When *CBDEV is low, the upper 12 of the 24 bits is high). If the maintenance register is being selected and the *CBWRT line is low, the maintenance register is loaded from the common bus. If the maintenance register is selected and the *CBWRT line is high, the contents of the maintenance register are sourced to the common bus.

There are 2 bits in the maintenance register that, when set, are automatically cleared by the hardware. These are STEPREQ (bit 12) and PCLKREQ (bit 11). When these bits are set, the clock performs a single bit or P clock respectively. Within the clock cycle, these clock request bits are cleared to insure the clocks go through exactly one cycle.

A third bit that has a type of auto clear is CBATTN (bit 15). This bit triggers the common bus state machine to perform a common bus cycle. At the end of the common bus cycle, this bit is auto cleared by the state machine.

For further details on maintenance register bit definitions, consult the PS 390 Raster Backend Microcode Manual.

NOTE: CBATTN should never be set by the JCP. When set, this bit triggers the Bitslice to perform an interrupt.

2.2.12 Common Bus Interface

The common bus interface supports interrupt generation to the JCP and Direct Memory Access (DMA) to Mass Memory.

2.2.12.1 Interrupts

The PS 390 Raster Backend primarily uses interrupts as a mechanism to transfer data, such as pick information, to the JCP. Before interrupting the JCP, the interrupt vector register must be loaded from the D-Bus. The interrupt is initiated by clearing the *DDIRV bit in the execution register. This bit is cleared by writing the jump vector to the upper 8 bits of the interrupt register on the D-Bus. This sets a 74F74 register. Before the interrupt can be forwarded to the JCP, the interrupt must be synchronized with the JCP clock. The synchronization is accomplished by running the output of the 74F74 to a second 74F74 which is clocked with the JCP clock. The second 74F74 generates a level 3 interrupt request for the JCP.

When the JCP is ready to handle the interrupt, it acknowledges the interrupt by setting the CBIIN[3] line and clearing the *CBWRT, *CBLDS and *CBADS lines. When this condition is detected by the Bitslice's CB/XREG PAL, the interrupt vector is put on the common bus data lines and a data acknowledge (DTACK) is issued to the JCP. When the JCP receives the interrupt vector, it lowers the CBIIN[3] line causing the CB/XREG PAL to raise the DTACK line, remove the interrupt vector from the common bus data lines and remove the interrupt request. This completes the interrupt cycle.

Note: The JCP card is equipped with a set of jumpers which are required for the JCP to service the interrupt. Also pins 21 & 22 on the P2 connector on the backplane of the PS 390 cabinet should be jumpered together in order to service the interrupt. (Refer to the JCP installation manual under interrupts.)

2.2.12.2 Direct Memory Access

The DMA provides the Bitslice with a means of directly accessing mass memory. The DMA logic is comprised of a common bus data register, common bus address register, common bus arbiter, time out logic and the DMA state machine.

A scenario of the sequence of events required to access mass memory is as follows. To write into mass memory: Load the common bus data register with the value to be written to mass memory. Load the common bus address register with the mass memory address to be written. Loading the high byte will trigger the DMA state machine. Therefore, the low address word must be loaded first. The DMA state machine will perform the necessary handshaking with

the JCP to gain control of the common bus and write the contents of the data register to mass memory. While the DMA state machine is performing the data transfer the CBBUSY bit in the status register will be high. Upon completion of the data transfer, the CBBUSY is cleared. A second common bus cycle may not be requested until the CBBUSY flag is cleared.

When retrieving data from mass memory, the sequence of events is identical to writing to mass memory with the following exceptions. No data need be written to the common bus data register before writing the common bus address register. The high byte of the common bus address register is loaded using the read mode command rather than the write mode command.

The DMA state machine works in conjunction with the arbiter PAL to conduct the handshaking which is necessary to perform a common bus cycle. The cycle is triggered when the microcode writes the high byte of the common bus address register. This causes the DMA state machine to issue a COMMON to the arbiter PAL. The arbiter PAL then lowers *MINENEXT and issues a common bus request (*CBREQ) to the JCP. When the JCP is ready to surrender control of the common bus, it issues a common bus grant (CBGIN). The arbiter then lowers *IGOTIT declaring control of the common bus. The common bus address register is sourced to the common bus address lines, the data is sourced to the common bus data lines if a write operation is in progress and the common bus control lines are asserted. These include address strobe, lower data strobe, upper data strobe, the read/write line and the common bus acknowledge.

The last signal to be asserted is the DTACK from the JCP. When writing to mass memory, DTACK indicates the data has been taken and the cycle can end. The common bus address and data registers are disabled from the common bus lines and the control signals are deasserted, ending the cycle.

When reading from mass memory, DTACK indicates the data is available. The DMA state machine latches the data in the common bus data register, disables the common bus address and data lines and deasserts the control signals, ending the common bus cycle.

The common bus data register is a pair of 74AS652 transceivers. One I/O port is connected to the D-Bus and the other to the common bus data lines. The common bus address register is comprised of three 74F374 registers. Their inputs are connected to the D-Bus and their outputs are connected to the common bus address lines. Their output enable is tied to the *IGOTIT line of common bus arbiter PAL. This line indicates the DMA state machine has control of the common bus and enables the address to the common bus. The DMA state machine is coded in a 16R8B PAL. The time out circuitry is a pair of 74LS393s which are 4 bit counters clocked from the 100ns Bitslice clock.

2.2.13 Video Intermediate Register

The video intermediate register is comprised of two 74ALS652 bidirectional transceivers. It passes data to a specified control register on the video card or reads the contents of a video control register back to the Bitslice. The execution register contains a field for register addressing and mode control. There are 4 register select lines, a read/write control line that determines if the specified control register is to be read or written and a mode control line. The register address field is shared between the FSBC and the video card. The mode control line determines if the specified address is intended for FSBC or the Video card.

2.2.14 Clocks

The Bitslice clocks are generated with a 16R8B PAL. The PAL is clocked by a 25ns clock and generates a 100ns clock for the Bitslice. The PAL outputs are gated with AM29823 registers which lend themselves to accurate generation and duplication of the clock signals.

There are 4 modes of operation supplied by the clock PAL. Normal mode, in which all the signals with the exception of *CLRREQ run through a normal cycle. Halt mode, which allows the 50ns raw clock, 100ns raw clock and IEN75CLK to run through a normal cycle while all other signals are halted. Shift Mode, which is a modified version of halt mode, allows PCLK and STROBE to run. In addition to those signals that run during halt mode, Single Step, allows all the signals to run through one cycle.

The various clock modes are selected via the maintenance register. There is a halt bit, single step request and PCLOCK request. The halt bit determines if the clocks are in normal mode or halt mode. Once the clocks are put in halt mode, a step request or PCLK request can be issued. When either a step request or PCLK request is given, the appropriate clock mode runs through a single cycle. During those cycles the clear request signal is asserted to clear all mode requests. This guarantees exactly one clock cycle will be executed.

Normal mode allows the Bitslice to cycle through the WCS executing microcode. Halt mode stops the entire machine. This allows the microcode to be loaded in the WCS, loading of the vector RAM and debugging of the Bitslice. A PCLK provides a simple means of loading the WCS. Each time a PCLK request is asserted, PCLK runs through one clock cycle clocking data into the WCS and incrementing the microsequencer's internal address counter. Single Step is useful in debugging allowing the microcode to be walked through while checking the status of various components of the Bitslice.

NOTE: A clock cycle is composed of 4 phases, phase 0 through phase 3. When the halt signal is asserted, the clocks complete the current cycle and stop in phase 0. This prevents any clocked components from hanging in an unknown state.

2.3 The Endpoint Pipeline

The PS 390 Graphics System Endpoint Pipeline is comprised of several custom VLSI chips, that are also used on the Shadowfax project. They include the Delta Calculator chip, the Depth Cue chip, the Divider chip, the FIFO Stack Bus Controller chips and the Pixel Processors. In addition to the Shadowfax VLSI chips there is a state machine that controls the transfer of data from the FIFO/Stack Bus controllers to the Pixel Processors.

A brief description of each chip follows. For a detailed discussion of the custom VLSI chips, refer to the Shadowfax VLSI engineering manual.

2.3.1 The Delta / Depth Cue Calculator (DDCC)

The DDCC is comprised of three Depth Cue VLSI chips, one Delta Calculator VLSI chip and two Divider VLSI chips.

The Delta Calculator VLSI chip receives X and Y values from the FSBC chip and using these values the Delta Calculator computes the slope of the line, the adjusted endpoints and the number of pixels in the line and other values and flags required by the Pixel Processors.

The Depth Cue chips provide the Pixel Processors with the RGB intensity information required to draw depth cued lines. The Depth Cue chips set Z data for the FSBC. There are three Depth Cue chips, one for each color Red, Green and Blue.

The Divider chip performs a bit serial binary division of two 34 bit operands. There are two divider chips in the DDCC, one for the Delta Calculator and the other for the Depth Cue Chips.

The Pixelpipe hardware is comprised of the following components:

1. The Input FIFO Stack Bus controller manager
2. The Input FIFO Stack Bus controller
3. The Delta Depth Cue Calculator
4. The Output FIFO Stack Bus Controllers
5. The Pixel Processor Array Loader
6. The Pixel Processors

The Input FIFO Stack Bus controller manager.

The Input FIFO Stack Bus controller manager loads endpoint information into registers that reside inside the Input FIFO Stack Bus Controller. The hardware consists of a register and state machine that resides in a PAL.

When the Bitslice receives an endpoint from the PLS, it first processes this endpoint information to output the correct data format used in the Pixel Pipeline. Then the Bitslice has to write the 32 bit X, Y, Z, W registers inside the Input FSBC.

Because the Bitslice can only write one 16 bit word at a time, there is a 32 bit register, which the Bitslice can load by first writing a 16 bit LSW and then a 16 bit MSW.

Whenever the Bitslice writes the MSW of the 32 bit FSBC data register, the INFSBC controller writes a register inside the FSBC. Therefore the Bitslice must wait until the *INFSBCREADY bit is asserted by the INFSBC controller before writing any registers. When the input FSBC is ready, the controller may write the register inside the FSBC. The transfer completes when the Bitslice asserts the acknowledge signal for the input FSBC. The INFSBC controller passes the acknowledge signal to the input FSBC and then returns to the idle state where the controller remains until the next transfer of data.

Registers inside the input FSBC are addressed with the signals INFSBCADR[0..2]. When the Bitslice writes the MSW of the 32 bit data register, it provides the FSBC register address in the VID/BCREG[0..2] field of the microword. This address is latched by the INFSBC controller, and passed on to the input FSBC in the next state.

The VIDFSBCR/*W signal from the microword is also used by the INFSBC controller. When asserted it causes the LSW of the 32 bit data word for the input FSBC to be cleared to zero's. This is used as a fast clear for the LSW so that the microcode does not have to clear the register by writing to it.

2.3.2 The Input FIFO Stack Bus Controller

The FIFO Stack Bus controller chip is used to control static RAM arrays for stacks and FIFO. In the PS 390 Graphics System, the Input FIFO Stack Bus controller is used as a bus controller and to convert the data from 32 bit parallel to 32 bit serial words for the Pixel Pipeline Delta Calc and Depth Cue chips.

For a detailed description of the FIFO Stack Bus Controller refer to the Shadowfax VLSI manual.

2.3.3 The Delta Depth Cue Calculator

The DDCC consists of the Delta Calculator chip, 3 Depth Cue chips and 2 divider chips. One divider is used by the Delta Calculator and one is used by the Depth Cue chip which computes intensity information for the color Red.

The Delta Calculator receives the Command/Status, X, Y, Z, W words from the input FSBC. The Depth Cue chips receive only the Command/Status word and the Z component.

The command word and the vector components are routed from the input FSBC to the Delta Calculator and Depth Cue chips via the INFSBCO[0..4] signals.

For a detailed description of the Delta Calculator and Depth Cue chips, refer to the Shadowfax VLSI manual.

2.3.4 The Output FIFO Stack Bus Controllers

The Output FIFO Stack Bus Controllers gather serial information from the Delta Calculator and Depth Cue Chips, and convert the serial data to a parallel format. They also alert the Pixel Processor Array Loader (PPALoader) that data is available in the internal output registers, so that it can be passed to the Pixel Processors or returned to the Bitslice.

The Endpoint FSBC collects serial data from the Delta Calculator, while the Color FSBC collects serial data from the Depth Cue Chips. The Command/Status word in the data specifies if the data should be passed to the Pixel Processors or to the Bitslice. If data is to be returned to the Bitslice then the signal ENDFSBCA/*B is asserted HIGH, if data needs to go to the Pixel Processors the ENDFSBCA/*B is asserted LOW. After data is taken from the registers the FSBC are acknowledged with the *OUTFSBCPACK signal. If an acknowledge signal is not received, eventually the FSBC will assert the ENDFSBCBUSY signal, indicating to upstream stages of the pipeline that no more data can be transferred down.

For a detailed description of the FIFO Stack Bus Controller refer to the Shadowfax VLSI manual.

2.3.5 The Pixel Processor Array Loader

The Pixel Processor Array Loader is a state machine that transfers data from the output FSBC's to the Pixel Processor Array or back to the Bitslice.

When the *OUTFSBCPREQ signal is asserted, the PPAloader transfers data to the Bitslice if the OUTFSBCA/*B is asserted HIGH, and transfers data to the Pixel Processor array if the OUTFSBCA/*B signal is asserted LOW.

During transfer of data to the Bitslice a 16 bit word is transferred from the FSBC to a data register, where the Bitslice can read it. After the Bitslice has read this word the PPAloader transfers the next word, and this process continues until the Bitslice asserts the PPLPACK signal, indicating that enough information has been transferred. Then the PPAloader asserts the acknowledge signal for the output FSBC's and returns to the idle state, where it remains until the next request from the Output FSBC.

When the transfer of data is to the Pixel Processors, the PPAloader must first wait until the Pixel Processors are ready to take data into their input registers. This is indicated when the *NPR (New Packet Request) signal from the Pixel Processors is asserted. The PPAloader transfers a word of data every 100 ns, until all the registers in the pixel processor's input stage have received new data. Then the PPAloader asserts the *PIXPACK signal indicating that the transfer of data is complete. The PPAloader then returns to the idle state where again it waits until a request is made by the output FSBC.

The hardware associated with the PPAloader includes the PPAloader state machine, a FSBC address generator, a Pixel Processor address generator and some additional conditioning logic to control the output enable signals for the FSBC's.

2.3.6 The Pixel Processors

The Pixel Processors draw anti-aliased lines based on endpoint and slope data received from the Delta/Depth Cue Calculator. The Pixel Processors also provide read and write access to the PS 390 Graphics System Frame Buffer.

The PS 390 Graphics System has a total of sixteen Pixel Processors, with eight processors on each Raster Backend Card. All memory access to the random port of the video RAMs is through the Pixel Processors. Each Pixel Processor has an array of video RAMs which it reads and writes. The Pixel Processors contain the counters and registers which hold the current row for both dynamic memory refresh cycles and screen refresh cycles.

The Pixel Processors respond to commands given them by the PPAloader or the Bitslice. The Pixel Processors are the only interface between the Frame Buffer and the other parts of the PS 390 system.

The Pixel Processors also transfer data via a two bit interface to the Pixel Read Machine. For a detailed description on the Pixel Read Machine refer to Section 2.3.7 of this manual.

The Pixel Processors are described in detail in the Shadowfax VLSI Engineering Manual.

2.3.7 The Pixel Read Machine

The task of the Pixel Read Machine is to retrieve pixel information from the Raster Backend Frame Buffer.

The Pixel Processors have a two bit data port through which pixel information may be retrieved. Since only two bits can be retrieved at any time, some mechanism has to be provided to select the appropriate bits from the 32 bit data register inside the pixel processor. The correct bits can be selected by providing an address on the Pixel Processor Address bus. As shown in Table 2.1, (2 bits) can be selectively retrieved.

The Pixel Read machine is implemented using two AMD 29818 registers, two 8 to 1 multiplexers, a state machine and some additional conditioning logic. The AMD29818 register has a onboard diagnostic shift register that can be loaded serially. The multiplexers serve to select the morsels from the correct pixel processor.

To obtain pixel information from the Frame Buffer, the Bitslice first writes the register that specifies from which pixel processor the read will occur. Then the Bitslice commands the Pixel Processors to perform a Scan Line Buffer Read cycle from the Frame Buffer. When the Pixel Processors perform this cycle the *SLR (Scan Line Request) signal is asserted. This causes the Pixel Read State Machine to assert the *PPWAITSLB signal, which halts the Pixel Processors. The Pixel Processors are halted until the wait signal is negated.

After the Pixel Processors are halted, the state machine shifts 16 bits into the diagnostic registers of the AMD29818's, then transfers these 16-bits to the pipeline register of the AMD29818. Again 16-bits are transferred from the pixel processor to the diagnostic register of the AMD 29818, and then the state machine asserts the *SLBREADY signal, indication to the Bitslice that the pixel data is ready to be read. When the Bitslice reads the first word from the AMD 29818 pipeline register, the state machine transfers then next word from the AMD 29818 diagnostic register to the pipeline register. The Bitslice again reads the pipeline register to retrieve the second word of pixel information, and then the Pixel Read state machine returns to the idle state, releasing Pixel Processors from their halt state.

2.4 The Frame Buffer

The Frame Buffer is located partially on the 204485 board which has the Bitslice processor, and partially on the 204486 board which has the Video Output section.

The PS 390 Graphics System Frame Buffer is very similar to the Shadowfax Frame Buffer. The Frame Buffer components include:

Table 2.1: Pixel Read Machine Morsel Select Bit Values

Color Select	*Address PPADR[5..0]*
Blue bits 0,1	b'000000'
Blue bits 2,3	b'000001'
Blue bits 4,5	b'000010'
Blue bits 6,7	b'000011'
Green bits 0,1	b'000100'
Green bits 2,3	b'000101'
Green bits 4,5	b'000110'
Green bits 6,7	b'000111'
Red bits 0,1	b'001000'
Red bits 2,3	b'001001'
Red bits 4,5	b'001010'
Red bits 6,7	b'001011'
Window bits 0,1	b'001100'
Window bits 2,3	b'001101'
Window bits 4,5	b'001110'
Window bits 6,7	b'001111'

- 1024 × 1024 × 48 Image bit planes.
- 1024 × 1024 × 4 Window planes.
- 1024 × 1024 × 2 Valid planes.
- Frame Buffer Memory Controller.
- Video Serial Port Interface

2.4.1 Video RAMs

The Frame Buffer uses Video RAMs as the storage element. The Video RAMs have a dual port architecture. The two ports are the random port and the serial port. The Pixel Processor Array accesses the Video RAMs through the random port. The Video RAM looks like a 256 × 256 × 4 array from this port.

The serial port of the Video RAMs is used to read out the video data. The serial port looks

like a 4 bit wide, 256 bit long, 40 MHz shift register. This shift register is written and read in a parallel fashion using the random port and special control signals. These operations are known as transfer cycles.

A fast clear of the Video RAM is accomplished by writing one row ($1 \times 256 \times 4$) to the desired value, transfering that row into the shift register, and then transfering the shift register into the other 255 rows. The entire Video RAM can be cleared to a known value with 512 operations. There are 256 random writes to clear the row, 1 transfer operation to write the shift register, and 255 transfer operations to write the other rows.

2.4.1.1 Valid Planes Storage in Video RAMs

The nature of the fast clear operation of Video RAMs requires the valid plane for the top half of the picture to be stored "next to" the valid plane for the bottom half of the picture. The valid plane for the other buffer is stored in the bottom half (rows $128 \rightarrow 255$) of the video RAM. The fast clear operation is as follows:

Row 0 is written to all 0 using random cycles ($1 \times 256 \times 2$).
Row 0 is transfered into the shift register.
The shift register is written into rows $1 \rightarrow 127$ ($128 \times 256 \times 2$).

In this way the contents of rows $128 \rightarrow 256$, which contain the valid bits for the other buffer, are not disturbed, and the clear is accomplished with only 374 operations.

2.4.2 Point Mapping of Pixel Processors & Video RAMs

Each processor has its own set of 14 Video RAMs, and is responsible for one sixteenth of the 1024×1024 Frame Buffer. The pixel processor is not responsible for a contiguous region, but is interleaved with the other fifteen pixel processors. Starting with the upper left hand corner of the screen, the mapping is as pictured in Figure 2.6. The numbers refer to pixel processor numbers.

2.4.3 Image Bit Planes

The 1024×1024 image bit planes are used to store the image being displayed. The 48 bit planes are divided up into two groups of 24 bits each. These groups make up the two buffers of the double-buffered image. One buffer is displayed while the other one is being updated. After the pixel processor array is done updating one buffer, the buffers are swapped, and the other buffer is updated while the first one is displayed.

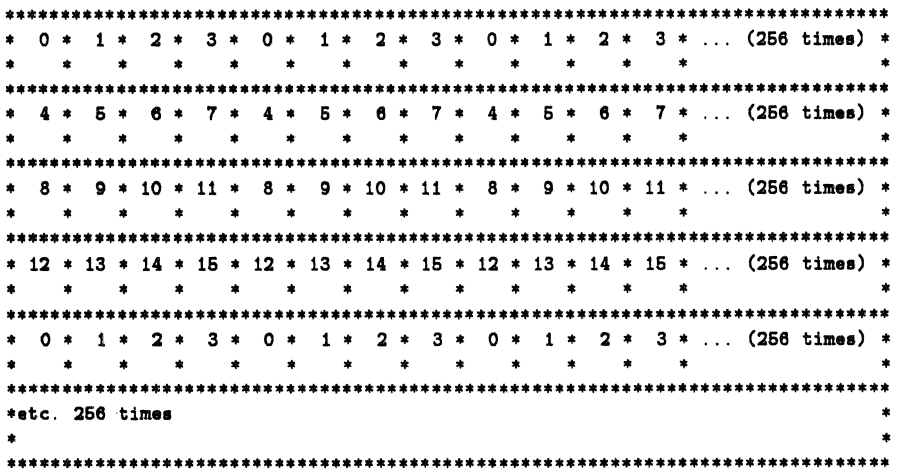


Figure 2.6: Pixel Mapping on the PS 390

The 24 bit planes in each buffer are divided up into three groups of eight bits each. The groups of eight bits are used to store the intensity for each of the primary colors red, green, and blue.

2.4.4 Window Planes

The four window bit planes are used for determining what type of image is stored in the image bit planes. The window numbers determine the attributes of each pixel. Window numbers zero through three are double buffered, non-blinking images. Window numbers four through seven are double buffered, blinking windows. The blink rate of these windows is controlled by the blink rate register. Window numbers eight through fifteen are single buffered, non-blinking windows. The characteristics of each window number are programmed into the window lookup PALs.

The window lookup PALs use the valid planes, window number, and the blink signal to tell the VLA to put out the input pixel data or background color.

2.4.5 Valid Planes

The two valid planes are used for fast screen clear. When the bit-slice swaps buffers, it clears out the valid plane of the buffer that the pixel processor array is about to start updating. As

the pixel processor array draws lines, it sets the valid bits where the lines are drawn. In double buffered windows, the video output section will display only the pixels where the valid planes are set. The background color will be displayed for pixels where the valid plane is not set. In single buffered windows the valid planes are ignored. The image should be written into both buffers for single buffered windows.

2.4.6 Frame Buffer Memory Controller

The Frame Buffer Memory Controller (FBMC) is the same as the Frame Buffer memory controller used on Shadowfax, with the exception that the PS 390 Frame Buffer memory controller does not support page mode access to the video RAMs.

The Frame Buffer Memory Controller (FBMC) provides timing to the Frame Buffer memory array and arbitrates requests for memory cycles. The FBMC provides RAS/CAS, Write Enable, and other memory timing signals to the Frame Buffer memory and the Pixel Processors. The Memory Controller is physically located on the Right Card (RC) of the Raster Backend.

The Memory Controller Arbitration and Next Cycle Control PAL receives requests for Frame Buffer memory cycles, conducts an arbitration and issues a 4-bit NEXTCYCLE code. NEXTCYCLE is presented to the Memory Cycle Sequencer PROM which tracks the cycle and generates a 6-bit CYCSTATE code every 100 ns. The CYCSTATE code is optimized to a 4-bit FBTST (Frame Buffer timing state) code which is sent to both Frame Buffer cards. The FBTST code is registered on the Frame Buffer card and is then presented to the TST Timing Pals which generate memory control signals every 25 ns. The CYCSTATE code is also interpreted on the Memory Controller by the Memory Cycle Acknowledge PAL which generates acknowledges and other control signals for use on the Memory Controller.

The FBMC handles ten distinct memory cycles:

- Reset
- Idle
- Dynamic RAM refresh
- Pixel Processor read
- Pixel Processor write
- Pixel Processor long read
- Pixel Processor long write
- Video shift register load cycle
- Fast clear read transfer cycle

- Fast clear write transfer cycle

The Reset cycle is executed continuously as long as the reset signal from the Bitslice is asserted. This results in Frame Buffer timing code that brings RAS and CAS up and holds them up.

In the Idle cycle, the Memory Controller is waiting for a memory cycle request.

The Dynamic RAM (DRAM) refresh cycle refreshes every row in Frame Buffer memory once every four milliseconds.

The Pixel Processor read cycle allows the Pixel Processors to read Frame Buffer memory. A cycle which is concluded after a single access is essentially a random cycle.

The Pixel Processor read cycle can only follow a PP read or write cycle which was concluded by bringing RAS up. Therefore, when the read cycle is begun, RAS falls. This cycle is required because the *ENDPPCYC0 signal bypasses the cycle sequencer PROM, leaving the PROM without knowledge of the condition of RAS.

The Pixel Processor writebreak cycle is similar to the readbreak cycle regarding pagemode, RAS, etc.

The Video shift register load cycle transfers a complete row from DRAM memory to the Serial Access Memory (SAM). This cycle occurs during every fourth horizontal retrace in non-interlaced modes, and every second horizontal retrace in interlaced modes. The Video Control card shifts the row out via the serial port during active horizontal time.

The fast clear read transfer cycle moves a row containing video from DRAM into SAM. This cycle is much like a Video Control shift register load cycle but is used to accommodate special control needed by the video memory and because the address select lines to the Pixel Processors (FAS0 and FAS1) are different for the two cycles.

The fast clear write transfer cycle writes the row back into DRAM 128 times, completing the clear operation.

2.4.6.1 Frame Buffer Memory Controller Signals

The Frame Buffer Memory Controller signals are defined as follows:

- Input Signals
  - Frame Buffer
  - Video Controller
- Output Signals

  - Slave Memory Controllers
- Internal Signals

2.4.6.1.1 FBMC Input Signals

***RBA (Read Bank A, Input, TTL)** *RBA indicates which bank of the Frame Buffer is read on a Pixel Processor read. This is a GEN signal.

PPFBMRQ0/2 (Pixel Processor on FB0/2 Memory Request, Input, TTL) PPFBMRQ0/2 indicates that a Pixel Processor on Bitslice C is making a request for a memory cycle. PPFBMRQ0/2 is created by ORing all eight memory requests from the Pixel Processors.

PPFBMRQ1/3 (Pixel Processor on FB1/3 Memory Request, Input, TTL) PPFBMRQ1/3 indicates that a Pixel Processor on the Video Card is requesting a memory cycle. PPFBMRQ1/3 is created by ORing all eight memory requests from the Pixel Processors.

READ (Read, Input, TTL) READ informs the Memory Controller that a Pixel Processor will perform a read cycle in two clock periods (200 ns), provided the Pixel Processors are not told to wait (FBWAIT or *SLBWAIT). When low, *READ indicates that a Pixel Processor will perform a write cycle. READ is a GEN pin.

2.4.6.1.2 Video Controller Input Signals

***PPSLBWAIT (*Pixel Processor/Scan Line Buffer Wait, Input, TTL)** *PPSLBWAIT indicates a wait caused by the Scan Line Buffer.

VCMREQ (Video Controller Memory Request, Input, TTL) VCMREQ informs the Memory Controller that the Video Controller is requesting a read transfer cycle to load four new screen refresh scan lines.

LASTTRN (Last Transfer Cycle, Input, TTL) LASTTRN informs the Memory Controller that the requested read transfer cycle is the last transfer of the current screen. This causes the Memory Controller to instruct the Pixel Processors to reset the screen refresh address counters.

2.4.6.1.3 FBMC Output Signals to Slave Memory Controllers

***FBALLCAS (Frame Buffer All Column Address Strobes, Output, TTL)** *FBALLCAS tells the CAS control logic to allow all memory arrays to execute CAS during memory cycle. This is required for BITBLTs and transfer cycles.

***FBRASA (*Frame Buffer Row Address Strobe Bank A, Output, TTL)** *FBRASA tells the RAS control logic to allow Bank A to perform RAS during a memory cycle.

***FBRASB (*Frame Buffer Row Address Strobe Bank A, Output, TTL)** *FBRASB tells the RAS control logic to allow Bank B to perform RAS during a memory cycle.

***FBRASW (*Frame Buffer Row Address Strobe Bank W, Output, TTL)** *FBRASW tells the RAS control logic to allow Bank W to perform RAS during a memory cycle. The Valid RAMs will always RAS even when this signal is not asserted.

***PPWAITFF (Pixel Processor Wait Flip-Flop, Output, TTL)** *PPWAITFF is set during the first 100 ns of a memory cycle (whether or not the cycle is a Pixel Processor cycle) if a Pixel Processor was making a request during the previous memory cycle. It is cleared during the last 100 ns of a Pixel Processor cycle.

FBFALD (Frame Buffer Address Load, Output, TTL) FBFALD tells the Pixel Processor to update the addresses in its memory refresh or screen refresh address registers as indicated by the state of FAS0 and FAS1.

FBFAS0 & FBFAS1 (Frame Buffer Address Select [0,1], Output, TTL) These two signals select the address output to the Frame Buffer from the Pixel Processors:

$$\begin{aligned}
&\text{FAS0} = 0,\ \text{FAS1} = 0 \Longrightarrow \text{line drawing address}\\
&\text{FAS0} = 0,\ \text{FAS1} = 1 \Longrightarrow \text{memory refresh address}\\
&\text{FAS0} = 1,\ \text{FAS1} = 0 \Longrightarrow \text{clear screen refresh address}\\
&\text{FAS0} = 1,\ \text{FAS1} = 1 \Longrightarrow \text{increment screen refresh address}
\end{aligned}$$

FBTST[0-3] (Frame Buffer Timing State, Output, TTL) These four signals indicate the current Frame Buffer timing state.

***PPWAIT (Pixel Processor Wait, Output, TTL)** *PPWAIT indicates that the Pixel Processors must wait due to some operation on the Frame Buffer card.

ENDPPCYC0 (End Pixel Processor Cycle, Output, TTL) ENDPPCYC0 is a signal from the arbitration and next cycle control PAL that tells the FST timing PALs on the Frame Buffer card to conclude the cycle.

PPMREQ (Pixel Processor Memory Cycle Request, Output, TTL) PPMREQ indicates that one of the 16 Pixel Processors is making a request for a memory cycle. The PPMREQ signal is created by ORing *PPFBMRQ0/2 and *PPFBMRQ1/3.

2.4.6.1.4 Internal Signals

RFMREQ (Refresh Memory Request, Internal, TTL) RFMREQ is a request made every 15.6 microseconds to refresh a new row of DRAM. This signal is generated by the refresh timer.

2.4.7 Register Description

The Memory Controller Mode Register determines the mode of the Memory Controller. It is a six-bit register which is one of the destinations of the Video Intermediate Register. It is write-only.

The bits are assigned as follows:

Bit #	*Name*
8	XFERCYCLE
9	*LONGCYCLE
10	RASBANKA
11	RASBANKB
12	RASBANKW
13	*PPRESET

XFERCYCLE set indicates that any cycles requested by the pixel processors will cause transfer cycles between the memory array and the shift register inside of the Video RAMs. This bit is used for the fast clear of valid planes when swapping buffers. This bit should be clear for line drawing operations.

***LONGCYCLE** cleared indicates that the memory cycle should be extended to accommodate slow Video RAMs. This bit is not used, and should be set for normal Video RAMs.

RASBANKA set indicates that pixel processor operations should affect the contents of Bank A in the Frame Buffer. If this bit is clear, the contents of Bank A will not be affected by the pixel processor operations.

RASBANKB set indicates that pixel processor operations should affect the contents of Bank B in the Frame Buffer. If this bit is clear, the contents of Bank B will not be affected by the pixel processor operations.

RASBANKW set indicates that pixel processor operations should affect the contents of Bank W in the Frame Buffer. If this bit is clear, the contents of Bank W will not be affected by the pixel processor operations.

***PPRESET** clear resets the entire pixel processor array. This bit should be set during normal operation.

2.4.8 Video Serial Port Interface

The Serial Port of the Frame Buffer Memory is under the control of the Video Timing Controller located on the Right Card of the Raster Backend. It runs asynchronous from the Random Port. Bank and Row control signals from the Video Control card cause the pixel and window data in the Video Ram Serial Access Memory (SAM) to be shifted out into the pixel pipe. Data is clocked out of the Video Ram at one-quarter pixel rate. Before leaving the Frame Buffer card, the pixel and window data is multiplexed to one-half the pixel rate.

2.4.9 Input/Output Interface Description

The Frame Buffer can only be written through the Pixel Processor Array.

The Frame Buffer can be read through the Pixel Read Machine or the video output section. The video output section can provide serial access to pixels, but this can only be used for diagnostic purposes.

2.4.10 RAS and CAS Control

Frame Buffer card RAS and CAS control is determined by the type of cycle and the condition of the signals RPRASBANKA, RPRASBANKB, RPRASBANKW and *RBA. The write mask is loaded at the fall of RAS. The Bitslice Processor writes RASBANKA, RASBANKB and RASBANKW into the FBMC mode register. *RBA is a Pixel Processor signal.

Frame Buffer memory is divided into sixteen arrays (one for each Pixel Processor) and four banks (Bank A, Bank B, Window Bank, and Valid Bank). CAS is controlled per array and RAS is controlled per bank.

Table A.38 shows the Frame Buffer RAS and CAS control.

2.4.11 Description of Maintenance Features

Problems with the Frame Buffer can be diagnosed by using the (Scan Line Buffer Pixel Read Machine) and the signature readback path in the video output. The visual characteristics of the display can be used very effectively to diagnose bad pixel processors and video RAMs.

2.5 The Video Output System

The PS 390 Graphics System Video Output System is similar to the design used in Shadowfax. The PS 390 Graphics System uses the Video Logic Array designed for Shadowfax. The major differences between the two designs are the lack of the writeable window lookup table on the PS 390, and the use of color lookup tables built into the digital to analog converters (DACs).

The hardware components of the Video Output System include:

- Master Bitslice / Video interface.
- Video Timing Controller.
- Automatic Blinking.
- Light Pen Support Hardware.
- Cursor Display Generation Hardware.
- Video Pixel Pipeline.
- Frame Buffer Interface.
- Window Lookup Table.
- Video Logic Array.

- Digital to Analog Converters (DACs).
- Pixel Signature Readback.

Figure 2.7 shows a block diagram for the PS 390 Video Controller.

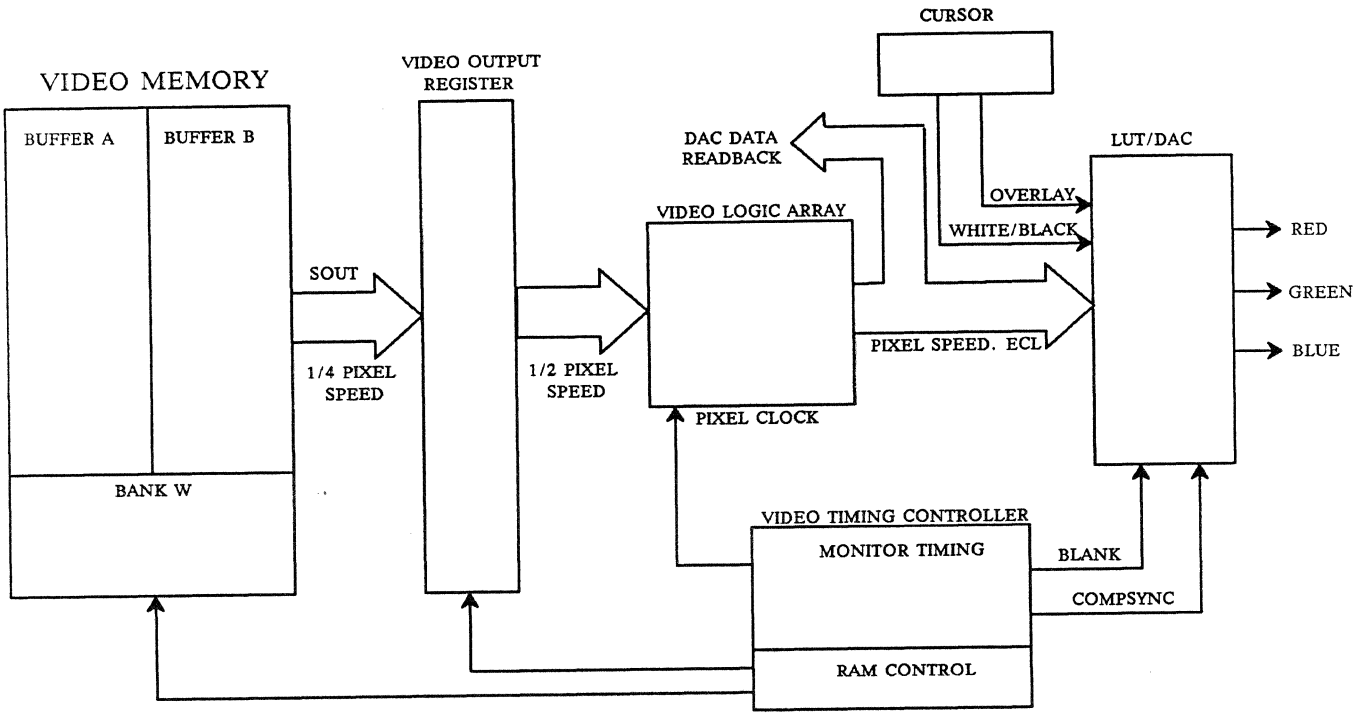


Figure 2.7: Block Diagram of the PS 390 Video Controller

2.5.1 Master Bitslice/Video Interface

The Master Bitslice/Video Interface is used to read and write the various control and status registers on the Bitslice and Video Output boards. The Master Bitslice/Video interface consists of the Video Intermediate Register, the video card registers, and the decoding logic to control which registers are written and read.

The standard mode of operation is that the master Bitslice writes the Video Intermediate Register during one state, then writes the contents of the Video Intermediate register into one of the video registers during the next state. The master Bitslice can write new data into the

Video Intermediate Register at the same time that it writes the old data into one of the video registers. The converse is true for reading registers. This allows a pipelining operation.

The VID/BCREG[0-3], VIDEO/*FSBC, and VIDFSBCR/*W bits control which register is to be written or read. These same bits are also used for writing to the FIFO Stack Bus Controller. VIDEO/*FSBC is in the logic HIGH state when reading or writing to the Video Output Card.

2.5.2 Video Timing Controller

The Video Timing Controller provides the correct timing signals for several different display devices. The controller supports three different display options: a high resolution format, a TV format, and a camera format. Other formats can be supported, but require replacement of several parts in the video controller circuitry.

The Video Timing Controller is basically a massive state machine with output signals to control the digital to analog converters and the Frame Buffer. The state machine runs off of VTCCLK, which runs at one fourth of the pixel frequency.

The video timing controller divides into three subsections. One controls horizontal timing, one controls vertical timing, and one controls the Frame Buffer.

2.5.2.1 Horizontal Timing Subsection

The backbone of the horizontal timing subsection is a nine bit counter which increments every four pixels. The count decode PALs are programmed to watch for the counts where a signal should change, and then send the proper signal to a bank of J → K flip-flops, which actually store the signals HBLANK, CMPSYNC, and PIXFLOWING. HBLANK indicates that the video should be at BLANK level for horizontal retrace. CMPSYNC is composite sync, which tells the monitor to begin a horizontal or vertical retrace. CMPSYNC is also used to clock the vertical timing subsection. PIXFLOWING is used to start and stop the pixel pipeline. At the end of every horizontal line the signal *CLRHORZCNTR from the count decode PALs sets the horizontal counter back to zero.

2.5.2.2 Vertical Timing Subsection

The vertical timing subsection is identical in concept to the horizontal timing subsection. It uses a 12 bit counter, and is clocked once every line by the signal CMPSYNC. The signals that are stored in the bank of J → K flip-flops are VBLANK, HMSVBLANK, VSYNC, VEQZ, and LINEGTE512. VBLANK, VSYNC, and VEQZ are control signals to the horizontal timing

subsection to tell it what kind of line to display. VBLANK indicates that the display should blank its video for vertical retrace. HMSVBLANK indicates to the master Bitslice that vertical retrace is happening. HMSVBLANK negates one horizontal line before the actual end of vertical blanking so that the master Bitslice will not start some operation and have it interrupted by the active video. In addition, HMSVBLANK can be cleared by the master Bitslice so that the master Bitslice can keep track of when end-of-frame processing has been done. VSYNC indicates to the display that it should begin vertical retrace. VEQZ indicates to the horizontal timing subsection that equalization is happening. VEQZ is used for interlaced displays and for strange lines on non-interlaced displays.

2.5.2.3 Frame Buffer Control

The Frame Buffer control subsection is a state machine which also runs off of CMPSYNC. This subsection produces the signals *EVENFLD, *ROWMSB, *ROWLSB, VCMREQ, and LASTTRN. *EVENFLD is asserted when using an interlaced video format to indicate that the present field is the even scan lines. *ROWMSB and *ROWLSB select which of the four rows of video RAMs should be output-enabled. The two signals are the two least significant bits of the display row address. VCMREQ is asserted to request a video transfer cycle from the Frame Buffer Memory Controller. This signal is asserted every four lines with non-interlaced formats, and every two lines with interlaced formats. LASTTRN is asserted to tell the Frame Buffer Memory Controller that the next rows transferred will be the last rows on this screen to be transferred. This signal is asserted one horizontal line before the last time VCMREQ is asserted to ensure that LASTTRN gets to the Frame Buffer Memory Controller before VCMREQ does.

2.5.3 Custom Video Formats

The PS 390 Video Timing Controller can be programmed to support different video formats other than the standard three which are programmed in a standard system. There are limitations on pixel frequency, number of pixels per line, and number of lines per frame.

The Vertical section of the Video Timing Controller can specify eight types of horizontal lines for the Horizontal section. The signals VBLANK, VSYNC, and VEQZ specify the type of line. The Vertical section is programmed to give the right number of each type of line in the frame. The Vertical section also controls how many lines are in the complete frame.

Non-interlaced formats use three types of lines:

1. *Active Video:* VBLANK, VSYNC, and VEQZ signals are not asserted.

2. *Vertical Retrace:* Only VBLANK is asserted.
3. *VSYNC:* VBLANK and VSYNC are asserted.

Interlaced formats use six types of lines:

1. *Active Video:* VSYNC, VEQZ, and VBLANK signals are not asserted.
2. *Vertical Retrace:* Only VBLANK is asserted.
3. *Equalization (Long SYNC):* Only VEQZ is asserted.
4. *Equalization:* VEQZ and VBLANK are asserted.
5. *Retrace (Short SYNC):* VSYNC and VBLANK are asserted.
6. *SYNC:* VBLANK, VEQZ, and VSYNC are asserted.

The Horizontal part of the video timing controller determines the characteristics of each type of line:

- When does COMPSYNC start?
- When does COMPSYNC end?
- When does HBLANK start?
- When does HBLANK end?
- When does the pixel pipe start?
- When does the pixel pipe end?
- How many pixels are in the line?

The timing of the signals can be programmed to a resolution of four pixels. Most video formats can be programmed into the timing controller by programming the right values for the preceding parameters. The pixel frequency is limited to 70 MHz.

2.5.3.1 Setting Up a Custom Video Format

Two examples are given of how to customize a video format: one non-interlaced format, and one interlaced format.

2.5.3.1.1 Example 1:

Format: DEC VR-290. 1024 × 864, 60 Hz, non-interlaced.

Step 1: Determine pixel frequency. The timing specification gives the active video time on each horizontal line as 14.81 μsec. 14.81 μsec/1024 pixels gives 14.452 nsec/pixel, or a pixel frequency of 69.1968 MHz. This is the maximum pixel frequency allowed.

Step 2: Build your horizontal line. The timing specification gives the horizontal period as 18.5 μsec. 18.5 μsec / 14.452 nsec /pixel gives 1280 pixels /line. By similar calculations the front porch, sync pulse, and back porch are found to have the following number of pixels in them:

Front Porch:	12 pixels	(160 nsec)
Sync Pulse:	128 pixels	(1850 nsec)
Back Porch:	116 pixels	(1680 nsec)

Total blanking time is the sum of these, or 256 pixels. There are 1024 pixels of video, for a total of 1280 pixels per line.

The Video Timing controller can only handle down to 4 pixel resolution, so the numbers must be rounded to the nearest multiple of four. After the calculations are done, check to see that the numbers still add up to one complete line.

Step 3: Program the counts. Take all of the pixel counts and divide by four. These are the counts that are needed for the Horizontal Decode PAL. The count is zero just after the beginning of COMPSYNC. The Vertical part of the timing controller changes at the beginning of COMPSYNC, so the line type changes at the beginning of COMPSYNC. The pixel pipe needs to be started and stopped 16 pixels (4 counts) before and after the end and beginning of HBLANK. With this information, the counts are as follows:

Video Line:

Count	*Action*
31	End COMPSYNC
56	Start Pixel Pipe
60	End HBLANK
312	Stop Pixel Pipe
316	Begin HBLANK
319	Clear Horizontal Counter, Begin COMPSYNC

VBLANK Line:

Count	*Action*
31	End COMPSYNC
60	End HBLANK
316	Begin HBLANK
319	Clear Horizontal Counter, Begin COMPSYNC

VSYNC Line:

Count	*Action*
60	End HBLANK
287	End COMPSYNC
316	Begin HBLANK
319	Clear Horizontal Counter, Begin COMPSYNC

Notice that COMPSYNC always begins at the same count. The horizontal oscillator in the monitor always triggers on the beginning of COMPSYNC, so the beginning of COMPSYNC cannot vary.

Step 4: Determine the vertical timing. On the VR290 the specification says vertical sync 3H, no vertical front porch, total blanking interval 37H. The vertical back porch must be 34H. Total vertical time is 901H. (864H video + 37H blanking) The vertical front and back porches are probably the best places to fudge the timing to give the desired frame rate.

Step 5: Program the counts. Several cautions are in order. The next frame officially begins at the beginning of vertical blank to allow as much time as possible for end-of-frame processing. HMSVBLANK ends one line earlier than actual VBLANK to allow the Bitslice to finish its end of frame processing. LINEEQ512 actually happens at line 511 because there is

another clock delay until LINEGTE512 asserts. LASTTRN needs to assert 1 line before the last actual transfer of the field to ensure that the LASTTRN signal gets to the Frame Buffer Memory Controller before the last transfer request.

Count	*Action*
2	End VSYNC
35	EARLYENDVBLANK
36	End VBLANK
548	LINEEQ512
895	LASTTRN
900	Start VBLANK, Start VSYNC, CLRVRTCNTR

2.5.3.1.2 Example 2:

Format: NTSC, 640 × 484 Interlaced, 30 Hz

With interlaced formats there is a requirement for equalization pulses at double the horizontal frequency for a period of time during the vertical blanking period. During the equalization period the horizontal frequency is doubled, giving a COMPSYNC start every 1/2 H in order to put out these equalization pulses. This also allows the Video Timing Controller to start VSYNC in the middle of a horizontal line between the first and second fields of the frame.

The RS-170 and RS-343 standards call for equalization pulses, 6 in number, both preceding and following the vertical sync pulse. This means that the blanking interval between the first and second fields has two kinds of lines in it. The first line after the active video is only 1/2H long, but has a sync pulse as wide as an active video line. The first full length line after the vertical sync pulse has sync pulse which is only as wide as the equalization pulse, even though the line is a full 1H long. These lines are obtained by starting VBLANK 1/2H later than the first line which is blanked, and by putting in a "false" extra vertical sync pulse 1H long after the equalization pulses end.

Step 1: Determine pixel frequency. The timing specification gives the active video time on each horizontal line as 52.456 μsec. 52.456 μsec/640 pixels gives 81.900 nsec/pixel, or a pixel frequency of 12.21 MHz. This is well within the maximum allowable pixel frequency.

Step 2: Build your horizontal line. The timing specification gives the horizontal period as 63.556 μsec. 63.556 μsec / 81.9 nsec / pixel gives 776 pixels /line. By similar calculations the front porch, sync pulse, and back porch are found to have the following number of pixels in them:

Front Porch:	20 pixels	(1638 nsec)
Sync Pulse:	60 pixels	(4914 nsec)
Back Porch:	56 pixels	(4586 nsec)

Total blanking time is the sum of these, or 136 pixels. There are 640 pixels of video, for a total of 776 pixels per line.

Step 3: Program the counts. Take all of the pixel counts and divide by four. These are the counts that are needed for the Horizontal Decode PAL. The count is zero just after the beginning of COMPSYNC.

Active Video:

Count	*Action*
14	End COMPSYNC
24	Start Pixel Pipe
28	End HBLANK
184	Stop Pixel Pipe
188	Begin HBLANK
193	Clear Horizontal Counter, Begin COMPSYNC

Vertical Retrace:

Count	*Action*
14	End COMPSYNC
188	Begin HBLANK
193	Clear Horizontal Counter, Begin COMPSYNC

Equalization (Long SYNC):

Count	*Action*
14	End COMPSYNC
96	Clear Horizontal Counter, Begin COMPSYNC

Equalization:

Count	*Action*
6	End COMPSYNC
96	Clear Horizontal Counter, Begin COMPSYNC

Retrace (Short SYNC):

Count	*Action*
6	End COMPSYNC
188	Begin HBLANK
193	Clear Horizontal Counter, Begin COMPSYNC

VSYNC:

Count	*Action*
81	End COMPSYNC
96	Clear Horizontal Counter, Begin COMPSYNC

Step 4: Determine the vertical timing. An interlaced frame consists of two fields, one of which displays the even scan lines, and one of which displays the odd scan lines. In the Video Timing Controller, the Odd field is the field which is displayed first. It contains the top scan line on the screen (Numbered 0 by the pixel processors). The half lines that are displayed with television pictures are not displayed by the Video Timing Controller.

The general scheme of the video format is a blanking interval followed by 242 lines of active video, followed by another blanking interval followed by 242 lines of active video. Remember that blanking intervals are not identical. The timing specification is as follows:

3H	Equalization
3H	VSYNC
3H	Equalization
11H	Vertical Retrace
242H	Active Video (Odd numbered scan lines)
.5H	Equalization (Long SYNC)
3H	Equalization
3H	VSYNC
2.5H	Equalization
1H	Retrace (Short SYNC)
11H	Vertical Retrace
242H	Active Video (Even numbered scan lines)

Note that the sum is 525H lines. All interlaced formats have an odd number of scan lines in each frame, with VSYNC beginning half way through a scan line between the first and second fields.

Step 5: Program the counts. Remember that during VEQZ and VSYNC the line counter counts twice each line. Notice that action LINEEQ512 never occurs. There are only 484 lines of video in this format. If there were more than 512 lines of active video the action would occur twice during the frame. LASTTRN needs to assert 1 line before the last actual transfer of the field to ensure that the LASTTRN signal gets to the Frame Buffer Memory Controller before the last transfer request. Because VBLANK does not start until one line after the last line actually displayed during the first field, LASTTRN needs to be two lines later than would be expected.

The counts come out as follows:

Count	*Action*
5	Start VSYNC
11	End VSYNC
17	End VEQZ
27	EARLYENDVBLANK
28	End VBLANK
269	LASTTRN
270	Begin VEQZ
271	Begin VBLANK
277	Begin VSYNC
283	End VSYNC
288	Begin VSYNC, End VEQZ
289	End VSYNC
299	EARLYENDVBLANK
300	End VBLANK
539	LASTTRN
542	Begin VBLANK, Begin VEQZ, CLRVRTCNTR

For more information, see Table 2.2, Table 2.3, and the source files for the Horizontal Decode PAL, the Vertical Decode PAL, and the Frame Information PAL.

2.5.4 Automatic Blinking

Automatic Blinking Hardware allows the master Bitslice to set a blink rate for certain windows so that they will blink on and off independent of any action on the part of the master Bitslice.

The automatic blinking hardware consists of an eight bit counter which clocks every VSYNC, and is loaded from the master Bitslice. The count which is loaded is the desired blink rate in fields on /off subtracted from hex 100. Non-blinking requires a special case condition: setting the *DISPBLINK bit in the video control register. Whenever the counter reaches the hex value FF, it toggles the bit *BLINKON, which goes into the window lookup table to enable and disable the display of blink windows. Whenever the blink rate register is written, the blink windows will be displayed until the next time the counter reaches hex FF.

Table 2.2: Vertical Timing of Video Formats

VR290		VCount	Count at Start	VSYNC	VEQZ	VBLANK
3H	Synchronization	3	900	1	0	1
34H	Back Porch	34	2	0	0	1
864H	Active Video	864	36	0	0	0
901H	Frame Time	901				
RS-170 (NTSC)						
1st Field:						
3H	Equalization	6	542	0	1	1
3H	Equalization	6	5	1	1	1
3H	Equalization	6	11	0	1	1
11H	Vertical Retrace	11	17	0	0	1
242H	Active Video	242	28	0	0	0
2nd Field:						
.5H	Equalization (Long SYNC)	1	270	0	1	0
3H	Equalization	6	271	0	1	1
3H	VSYNC	6	277	1	1	1
2.5H	Equalization	5	283	0	1	1
1H	Retrace (Short SYNC)	1	288	1	0	1
11H	Vertical Retrace	11	289	0	0	1
242H	Active Video	242	300	0	0	0
525H	Frame Time	543				
RS-343 1024 × 864						
1st Field:						
3H	Equalization	6	956	0	1	1
3H	VSYNC	6	5	1	1	1
3H	Equalization	6	11	0	1	1
28H	Vertical Retrace	28	17	0	0	1
432H	Active Video	432	45	0	0	0
2nd Field:						
.5H	Equalization (Long SYNC)	1	477	0	1	0
3H	Equalization	6	478	0	1	1
3H	VSYNC	6	484	1	1	1
2.5H	Equalization	5	490	0	1	1
1H	Retrace (Short SYNC)	1	495	1	0	1
28H	Vertical Retrace	28	496	0	0	1
432H	Active Video	432	524	0	0	0
939H	Frame Time	957				

Table 2.3: Video Format Line Types

VSYNC	*VEQZ*	*VBLANK*	*Line Type*
0	0	0	Active Video
0	0	1	Vertical Retrace
0	1	0	Equalization (Long SYNC)
0	1	1	Equalization
1	0	0	Unused
1	0	1	Retrace (Short SYNC)
1	1	0	Unused
1	1	1	VSYNC

2.5.5 Light Pen Support Hardware

The light pen support hardware provides +5 volt fused power on the I/O panel for a light pen, and provides for latching of the screen address at the time of the light pen hit. The position of the tip switch is also readable.

When a light pen hit occurs, the TTL level pick signal comes onto the board, is converted to ECL, and then passed through a chain of six D flip-flops to synchronize the hit signal to the pixel clock. The synchronized signal %SYNLPPICK is used to register the ECL two pixel and four pixel clocks to get the single pixel resolution on the hit. The signal %4PLPPICK is produced by clocking %SYNLPPICK with the 4 pixel clock. The TTL version of %4PLPPICK is used to latch in the current horizontal and vertical counts from the video timing controller. The latched count can be read by the master Bitslice. There is no hardware in the video output system for automatic light pen tracking.

There are four bits in the video control register which can be used by the master Bitslice to do a screen blast if the light pen has lost the tracking cross.

2.5.6 Cursor Display Generation Hardware

Cursor display generation hardware allows a overlaying cursor to be displayed on the screen. The cursor can go off any edge of the screen. The cursor image contains three colors, one of which is controlled by a mask register.

The cursor display generation hardware can display one of two selectable cursor definitions. Both cursor definitions are writeable from the master Bitslice. The definition

allows for three colors for each pixel. The colors are clear, black, and mask. The mask color is controlled by the contents of the video control register. Clear allows the user to see through that portion of the cursor to the actual graphics. The colors that are displayed are ten percent brighter than the brightest color normally displayed, so a white cursor will be visible even on a white background.

Two sets of comparators continually monitor the counts in the vertical and horizontal timing subsections. When the vertical comparator detects that the cursor display should begin on the present line, it enables the horizontal comparator to detect the start of the cursor in the horizontal direction. The horizontal comparator detects the start of the cursor to only an eight pixel resolution. The horizontal start from the comparator allows an ECL down counter to start counting. When the ECL counter reaches zero, the cursor actually begins to be displayed. The ECL counter increments the cursor address counters every eight pixels, which brings another eight pixels into the 8177 Video Shift Registers.

The cursor can be made to go off of the top of the screen by placing the cursor at the top of the screen, and then changing the contents of the cursor Y start register so that display of the cursor begins part way through the definition of the cursor. No special treatment is required to make the cursor go off any other edges of the screen.

2.5.7 Pixel Pipeline

The pixel pipeline receives data from the Frame Buffer, and sends it to the data inputs to the DAC. It is converted to ECL and speed up to the pixel rate before it gets to the DACs.

The pixel pipeline consists of the pixel clock generation, the Frame Buffer interface, the pixel data paths, the window lookup table, and the video logic array.

The speed of the pixel pipeline is controlled by the pixel clock. The pixel clock is selectable from the video control register. The four clocks available are %HIRES, %DIAG, %CAM, and %TV. %HIRES is used for a high resolution display. %DIAG is used for diagnostic purposes. The actual signal is a bit in the video control register. %CAM is used for a color camera display format. %TV is used to drive a display with television timing.

The pixel clock is used to clock a counter which gives the 2 pixel, 4 pixel, and eight pixel clocks. The video RAM shift clocks are generated by combining the 4 pixel clock with delayed versions of the signal PIXFLOWING.

The pixel data comes out of the video RAMs 4 pixels at time, each at one quarter of the pixel speed. The data are registered in a bank of 74F399 registered multiplexers. The data coming from these registers is registered in the video output system in the 0/2 and

1/3 FBC registers. The pixel pipeline at this point is two pixels wide and runs at one half of pixel speed. The pixel color data from the FBC registers go directly into the video logic array. The pixel window data go into the color lookup table. The output from the color lookup table is one pixel wide and runs at pixel frequency. The output from the color lookup table connects into the video logic array. The output from the video logic array is one pixel wide and runs at pixel speed. The digital to analog converter inputs are connected directly to the video logic array.

2.5.8 Frame Buffer Interface

The Frame Buffer Interface requests video transfer cycles from the Frame Buffer Memory Controller, controls which video RAM outputs are sent to the pixel pipeline, and controls the rate at which the pixel data is sent.

The interface between the Frame Buffer and the video output system is used to send pixel data to the video output system from the Frame Buffer. The control signals are as follows:

- C0BABR0: Shift clock for column 0, banks A and B, row 0.
- C0BABR1: Shift clock for column 0, banks A and B, row 1.
- C0BABR2: Shift clock for column 0, banks A and B, row 2.
- C0BABR3: Shift clock for column 0, banks A and B, row 3.
- C0BWR0: Shift clock for column 0, bank W, row 0.
- C0BWR1: Shift clock for column 0, bank W, row 1.
- C0BWR2: Shift clock for column 0, bank W, row 2.
- C0BWR3: Shift clock for column 0, bank W, row 3.
- *RCLK0: Two pixel clock used to register the shift clocks on the Frame Buffer.
- SELCOL0: Select input to the 74F399 multiplexing registers on the frame buffer.
- *ROWLSB: Least significant bit of row address, used to decode which video RAMs to output enable.
- *ROWLSB: Second least significant bit of row address, used to decode which video RAMs to output enable.
- *BANKA: Frame Buffer bank A is being displayed, used to decode which video RAMs to output enable.
- *BANKB: Frame Buffer bank B is being displayed, used to decode which video RAMs to output enable.

- VCMREQ: Request to the Frame Buffer to transfer the next four rows into the video shift registers.
- LASTTRN: Indicates that the next transfer request will be the last transfer for this field.

2.5.9 Window Lookup Table

The window lookup table controls the display mode of the 16 windows which can be used. The window lookup table is not writeable.

The window lookup table is implemented in two 16R8B PALs run in parallel at one half of pixel frequency. The PALs receive data from the FBC registers, and send formatting information to the video logic array. Additional inputs to the PALs specify which of the valid bits to use (LINEGTE512) and if blink windows should be displayed (*BLINKON).

The assignment of window numbers is as follows:

- Windows 0 → 3: Double buffered, non-blink. Display pixel data if valid bit is set, otherwise display background color.
- Windows 4 → 7: Double buffered, blink. Display pixel data only if valid bit is set and *BLINKON is asserted, otherwise display background color.
- Windows 8 → 15: Single buffered. Always display pixel data.

The data from the window lookup PALs are converted to ECL, registered, multiplexed to pixel speed, registered, and sent to the video logic array. The total delay through the window lookup table is four pixels.

2.5.10 Video Logic Array

The video logic array on the video output system is used to convert the pixel data to ECL, speed multiplex it up to full pixel speed, and substitute in background color for invalid pixels. It is also used to load the color lookup tables in the digital to analog converters. In order to load the color lookup tables, the video logic array is taken out of display mode. The video logic array control register is loaded with E0 hex, which causes the video logic array to pass data directly from the color lookup table address register to the address inputs of the digital to analog converters. The counters inside the video logic array are not used.

2.5.11 Digital to Analog Converters (DACs)

The digital to analog converters take digital data at ECL logic levels and convert the data to analog signals used to drive the display. The digital to analog converters contain 256*8 lookup tables which are used for gamma correction. The lookup tables are accessible from the master Bitslice. The digital to analog converters produce video output signals at RS-343 standard levels.

The digital to analog converters are AM8151 Graphics Color Palettes. There are a total of three used, one each for red, green, and blue video. The green digital to analog converter is used also for composite sync. Only the horizontal sync input is used on the green DAC, because composite sync cannot be generated properly by the internal XORing of horizontal sync and vertical sync in the DAC. The video timing controller generates the proper composite sync signal.

Each DAC has a reference voltage and a reference current to control the output level. There is a trimpot which needs to be set to give an adjustment voltage of 1.076 volts. The adjustment voltages are located at E3, E4, and E5.

The AM8151s are specified to directly drive 50 ohm and 75 ohm RS-343 monitors. The monitor should have a 75 ohm termination on the video inputs. If more than one monitor is to be driven, they should be daisy-chained together, with only the last monitor in the chain having the 75 ohm termination on the video outputs. If large numbers of monitors are to be driven from one PS 390, a separate video buffer/amplifier should be used.

2.5.12 Pixel Signature Readback

The pixel signature readback hardware can be used to diagnose problems with the Frame Buffer and pixel pipeline. The master Bitslice can read the data which go to the digital to analog converters using the pixel signature readback hardware.

The actual data which are sent to the digital to analog converters can be read by the Master Bitslice through the pixel signature readback. This feature is not useful during runtime, but can be a powerful diagnostic tool. In order to use this feature, the display format should be set to diagnostic. Diagnostic format is high resolution timing at the speed controlled by the DIAGCLK bit in the video control register. The video system should be clocked until the end of vertical blank. The pixel data will begin coming to the DACs a fixed number of clock ticks after that. The data can be compared to the data in the Frame Buffer to check to see if the pixel pipeline is functioning properly.

2.5.13 Register Description

The registers include:

- Video Control Register: 16-Bits.
  Video Enable: 1 bit.
  Buffer Select: 1 bit.
  Video Format: 2 bits.
  Cursor Enable: 1 bit.
  Cursor Select: 1 bit.
  Vertical Blank: 1 bit. (Read-Only)
  Even Field: 1 bit. (Read-Only)
  Screen Blast: 4 bits. Enable and 3 color bits.
  Tip Switch Position: 1 bit. (Write 0 to clear hit registers).
- Background Color Register: 24 bits.
- Light Pen Hit X: 11 bits. (Read-Only)
- Light Pen Hit Y: 12 bits. (Read-Only)
- Cursor X Position: 11 bits.
- Cursor Y Position: 12 bits. (Double Register)
- Cursor RAM Address Register: 9 bits. Six bits are Cursor Row Start. (Write-Only)
- Color Lookup Table Address Register: 8 bits.
- Color Lookup Table Data: 11 bits. 8 bits data, 3 bits mask.
- Cursor RAM Overlay/Red data: 16-Bits.
- Cursor RAM Green/Blue data: 16-Bits.

2.5.14 Description of Maintenance Features

Most of the registers that are writeable are also readable. Some notable exceptions are the background color registers, the color lookup table address register, and the cursor RAM address register. The background color registers and color lookup table address register can be read through the signature path, and the cursor RAM address register can be tested by means of an address lines test similar to that used for the Mass Memory. In general most of the problems can be seen on the screen, and only minimal experience is needed to recognize the common problems.

One useful diagnostic feature of the video logic array is the fast write of lookup table mode. If the video logic array is not in display mode, the color tables have already been loaded, and screen blast is disabled, writing hex 40 to the video logic array control register will cause the video logic array to cycle through the color lookup table addresses, which

will display some rising gray scales on the screen. This test verifies that the video timing controller, video logic array, and digital to analog converters are functioning properly.

2.6 The Raster Display

The Raster Display or Monitor for the PS 390 is the FIMI 2054. The monitor is part of the PS 390 hardware and is technically not part of the peripheral set. However a brief introduction to its features may be useful.

The PS 390 can use either a FIMI 2054 or a DEC VR290 for displaying its raster images. The FIMI Monitor has a 20 inch viewing area while the DEC monitor offers a 19 inch veiwing area. Both monitors operate with AC 110/200 VAC, 60/50 Hz and a nominal power consumption of 150 watts. They have the following features:

Contrast Lets the user adjust the video display to a suitable intensity.

Brightness Lets the user adjust the background intensity to compensate for ambient room light.

Degauss Permits the user to clear color picture distortion caused by external magnetic interference.

Power On/Off Turns the monitor on and off. The monitor should be turned off at the end of the workday to extend its life.

Tilt Lock Locks or unlocks the tilting mechanism to allow or prevent movement; swivel operation is not affected.

These adjustment devices are located on the lower righthand side of the monitor.

Chapter 3

The PS 390 Peripheral Set

The PS 390 supports a variety of peripheral set configurations. With the initial release of the system, only the PS 300 peripherals can be supported. This includes the option of using either the LED or No/LED keyboards. The PS 300 peripherals require their own Peripheral Multiplexer.

Table 3.1: PS 300/Low Cost Peripheral Configurations

PS 300 Peripheral Multiplexer	*Low Cost Peripheral Multiplexer*
LED Keyboard No/LED Keyboard	VT220 Style Keyboard Switchable Dual Function (IBM & DEC) Keyboard
Function Button Array *(Optional)*	Function Button Array *(Optional)*
Interactive Control Dials *(Optional)* PS 300 Version with LEDs	Interactive Control Dials *(Optional)* Low Cost - No LEDs
Data Tablet *(Optional)* 6 × 6 Tablet 12 × 12 Tablet	Data Tablet *(Optional)* 6 × 6 Tablet 12 × 12 Tablet
Optical Mouse *(Optional)*	Optical Mouse *(Optional)*

The Peripheral Multiplexer combines the signals from the peripherals or interactive devices and transmits them to the PS 390.

The Low Cost Peripheral Set (which is the standard peripheral set for the PS 390) has its own Peripheral Multiplexer, and an option for using either a VT-220 style keyboard (Standard) or a Switchable Dual Function (IBM & DEC) keyboard (Optional), in addi-

tion to the other low cost peripherals.

The 32-Key Function Button Array is an extension of the programmable function keys on the keyboard. The keys are user definable.

The Control Dials have eight rotary encoders attached to dials which allow the user to manipulate screen images interactively.

The Data Tablets allow the user to use a stylus or cursor to encode position data into the PS 390.

The Optical Mouse sends x and y position data to the PS 390.

Figure 3.1 shows the PS 390 Graphics System and its associated peripheral devices.

3.1 The Peripheral Multiplexer

There are two Peripheral Multiplexers (or MUX Boxes) available for use with the PS 390. The first MUX box is for use with the PS 300 peripheral set. The second is designed to accommodate the new low cost peripheral set.

The Peripheral Multiplexer is a stand-alone metal box which is designed to fit beneath and form a pedestal for the Raster Display (FIMI 2054, DEC VR290, or equivalent). It furnishes the power to drive the peripherals and is also their point of connection to the system. Five connectors are provided on the front of the multiplexer which allows the various interactive peripherals to be connected to the system. Each connector is uniquely dedicated to the specific peripheral for which it is intended. Therefore, only one of each type of peripheral is allowed to be attached. The pinouts for these connectors are listed in Tables 3.2 and 3.3.

The Peripheral Multiplexer provides programmed logic which allows the data from interactive peripherals to be multiplexed over a single RS-232C line into the controller via one of the available communication ports.

Figure 3.2 shows the backside connectors and plugs for both Peripheral Multiplexers. Figure 3.3 shows the peripheral connections for both the PS 300 peripherals and for the low cost peripheral set. (Note the difference in the keyboard connectors.)

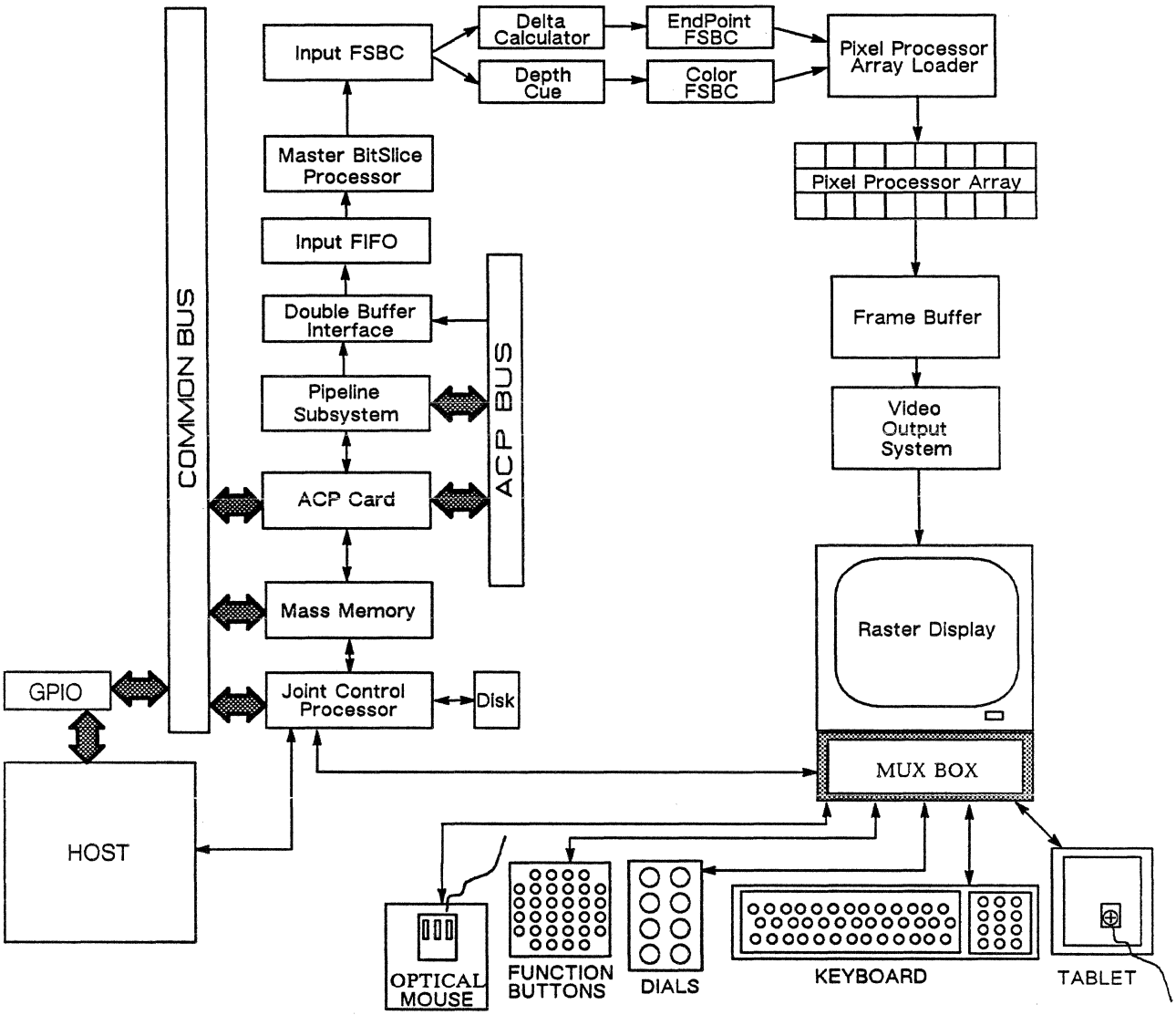


Figure 3.1: The PS 390 and Peripheral Devices

Table 3.2: Pin Assignments for the PS 300 Peripherals Multiplexer

Designation	*Connector*	*Pinout*
32-Key Buttons Dials Keyboard	RJ-11 90 degree PC Mount	1...+12 VDC 2...Ground 3...RS-422 OUT B 4...RS-422 IN B 5...RS-422 IN A 6...RS-422 OUT A 7...Ground 8...+12 VDC
Data Tablet Mouse	7-Pin Micro DIN (Hosiden)	1...Signal & Power Ground 2...Rec. Data from Device 3...Xmit Data to Device 4...-12 VDC for RS-232C 5...+5 VDC 6...+12 VDC 7...Device Present Shell - Chassis Ground
Graphics Controller	Amphenol 25 Pin D 117-DBMM-25SA (Military Socket including locking screw assembly)	1...Chassis Ground 2...Transmitted Data 3...Received Data 7...Signal Ground Other Pins not used

Table 3.3: Pin Assignments for the Low Cost Peripherals Multiplexer

Designation	*Connector*	*Pinout*
32-Key Buttons Dials	DuPont 68290-101 (2 rows of 4 pins)	1...+5 VDC 2...Ground 3...Device Present 4...Unused 5...+12 VDC 6...-12 VDC 7...TX Data 8...RX Data E.S.D. Shield-Chassis Ground
Keyboard	5 Pin DIN (5 contacts at 180 Degree Socket)	1...Signal Ground 2...Data In 3...Data Out 4...Signal Ground 5...+5 VDC Shell - Chassis Ground
Data Tablet Mouse	7-Pin Micro DIN (Hosiden)	1...Signal & Power Ground 2...Rec. Data from Device 3...Xmit Data to Device 4...-12 VDC for RS-232C 5...+5 VDC 6...+12 VDC 7...Device Present Shell - Chassis Ground
Graphics Controller	Amphenol 25 Pin D 117-DBMM-25SA (Military Socket including locking screw assembly)	1...Chassis Ground 2...Transmitted Data 3...Received Data 7...Signal Ground Other Pins not used

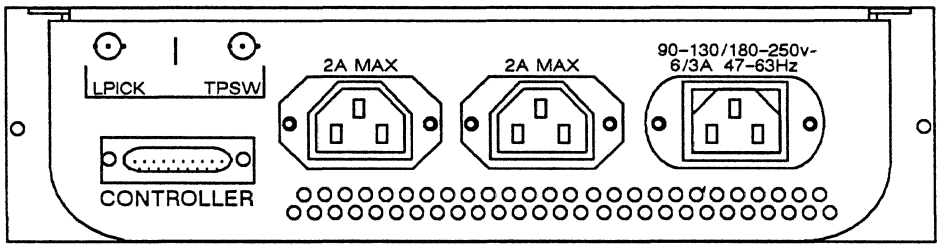


Figure 3.2: Backside Connectors for the Peripheral Multiplexers

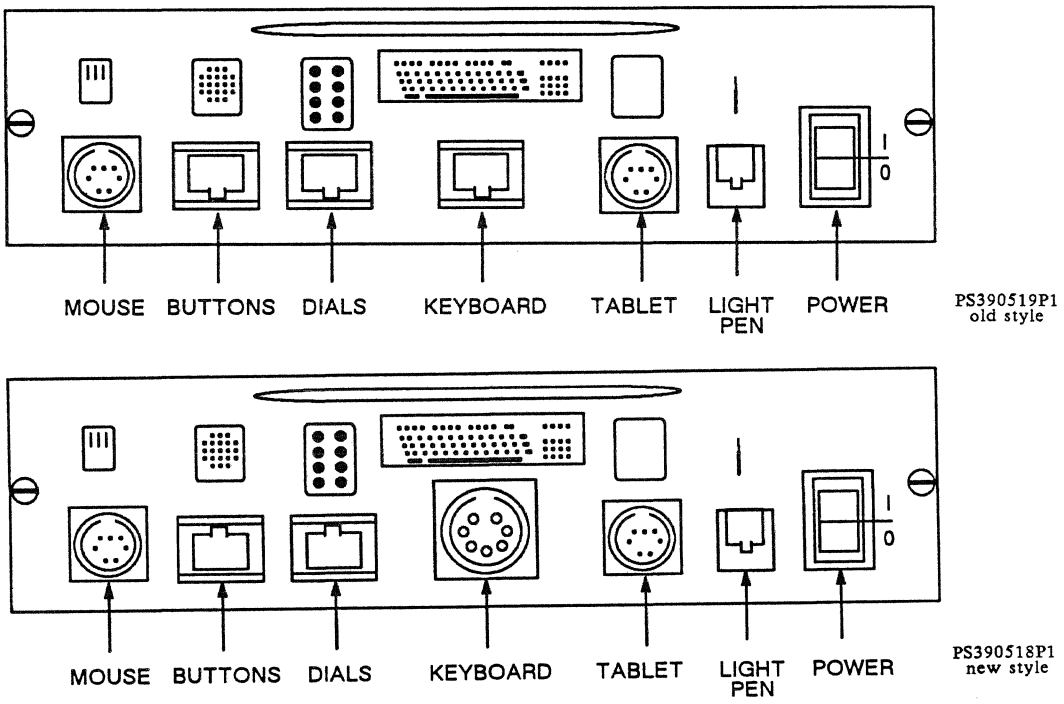


Figure 3.3: Multiplexer Connectors for the PS 300 and Low Cost Peripherals

3.1.1 Functional Characteristics

The Peripheral Multiplexer or MUX consists of a circuit card to which six input ports and one output port in addition to the input power port are connected. The six input ports support the following peripheral devices:

- The Keyboard
- The Control Dials
- A 32-Key Function Button Array
- A Data Tablet
- Spare Port for a Mouse or other device
- A Light Pen (Not Supported)

The function of the MUX is to receive input data from the peripheral devices and to multiplex the data onto an RS-232C output port to the PS 390, and to accept the data from the terminal controller and de-multiplex it to the various peripheral devices which are connected to it.

3.1.1.1 PS 300 Peripheral Set Device Addressing

The MUX has five active input ports on the front of the box, 1 output port on the rear of the box and 3 power connectors on the rear of the box. Device addressing assigns numbers to the ports. When one faces the front of the box the leftmost connector is a 7-Pin Micro DIN. It is for connection of a mouse when one is used. This port is addressed as port x'B4'. The next three connectors toward the right are RJ-11 connectors. The leftmost of these connectors is for the 32-Key Lighted Function Buttons and is addressed as port x'B3'. The next connector is for the control dials and is addressed as port x'B2'. The rightmost RJ-11 connector is for the keyboard and is addressed as port x'B1'. The 7-Pin Micro DIN connector at the right of the front panel is for the data tablet and is addressed as port x'B6'. When control messages are sent to the MUX box itself, they are addressed to port x'B0'.

3.1.1.2 Low Cost Peripheral Set Device Addressing

The MUX has five active input ports on the front of the box, 1 output port on the rear of the box and 3 power connectors on the rear of the box. Device addressing assigns numbers to the ports. When one faces the front of the box the leftmost connector is a 7-Pin Micro DIN. It is for connection of a mouse when one is used. This port is addressed as port x'B4'. The next two connectors toward the right are "Latch-N-Lock" connectors. The

leftmost of these connectors is for the 32-Key Lighted Function Buttons and is addressed as port x'B3'. The rightmost one of these DuPont connectors is for the control dials and is addressed as port x'B2'. The five-pin DIN connector is for the keyboard and is addressed as port x'B1'. The 7-Pin Micro DIN connector at the right of the front panel is for the data tablet and is addressed as port x'B6'. When control messages are sent to the MUX box itself, they are addressed to port x'B0'. These addresses are used for software compatibility with previous systems.

3.1.1.3 Light Pen

The PS 390 does not presently support the use of a Light Pen. A connector has been provided on the Peripheral Multiplexer to accommodate one in the future. At present, however, a Light Pen is not planned for the PS 390.

3.1.2 Transmission Characteristics

3.1.2.1 Multiplexing and De-Multiplexing

When peripheral data is received from an input port, the MUX queues the data to the output port for transmission to the host. If the last byte sent is from the same input device, then the MUX sends just that queued byte unless it is in the range of x'B0' to x'BF'. In that case it prefixes the queued byte with another byte with the value of x'B7'. If the data is from a different input device, then the MUX prefixes that byte only with the address of the device port, i.e., a value in the range of x'B0' to x'B6'. The MUX does not send another device port address until the source of the n+1st data byte differs from the source of the nth data byte.

When data is received from the host or terminal controller, the MUX sends the data byte received to the last addressed peripheral device until it receives a new port address in the range x'B0' thru x'B6'. It then sends all subsequently received data to that address until it receives a new address.

3.1.2.2 Flow Control

The flow of data to the peripheral devices is shut down when the MUX sends the sequence x'B030' to the PS 390. This sequence is the equivalent of "XOFF" in asynchronous protocols and is used to protect against the overrun of MUX buffers. When the MUX resumes receiving data from the PS 390, it sends the sequence x'B040'. This sequence is the equivalent to "XON" in asynchronous protocols.

3.1.2.3 Data Framing and Transmission Rates

The data sent to and from the MUX is asynchronous data with each byte containing eight data bits without parity plus one start bit and one stop bit. The data transmission rate of the MUX to and from the PS 390 is 19,200 baud. The transmission rates between the various peripherals and the MUX are shown in Table 3.4.

Table 3.4: Peripheral Device Transmission Rates

Device	*Baud Rate*
Keyboard Port x'B1' with Standard Keyboard	1200 Baud
" " with Dual Function Keyboard	9600 Baud
Control Dials Port x'B2'	9600 Baud
32 Func. Buttons Port x'B3'	9600 Baud
Mouse Port x'B4'	9600 Baud
Data Tablet Port x'B6'	9600 Baud

3.1.3 Diagnostic Loopback

The MUX will respond with the sequence x'B060' whenever it receives the sequence x'B0B9' from the PS 390. This response informs the PS 390 that the MUX is powered on and working, (meaning that it will recognize commands and respond to them).

3.2 The PS 390 Keyboards

The PS 390 can use three different keyboards:

(a) The standard PS 390 Keyboard is a DEC VT-220 style keyboard with a few minor modifications.

(b) The Dual Function Switchable Keyboard which is a 5085 Model 2 look-alike keyboard.

(c) The PS 300 Keyboards: E&S #204201-100 with LEDs and E&S #204201-101 without LEDs.

The PS 300 keyboard has its own peripheral multiplexer to accommodate it. PS 300 keyboard operation is described in detail in the PS 300 Document Set Volumes 1 & 5.

The VT-220 look-alike keyboard and the Dual Function Switchable keyboard must plug into the peripheral multiplexer which supports the low cost peripheral set.

Both keyboards meet safety and EMI/ESD qualifications.

Neither of the keyboards have LEDs for labeling of the function keys.

The PS 390 Keyboards main function is the generation and transmission of ASCII displayable characters, ASCII control characters, and PS 390 system sequences.

The Dual Function Switchable keyboard is also capable of having its output data stream interpreted as EBCDIC data (which is its principal mode of operation). These data are transmitted serially to the peripheral multiplexer. The transmitted data specifies displayed characters, commands, menu/table selections, etc.

The only operator controls located on the keyboard are the 95 keys used for data input.

The assembled keyboard measures 21.1 inches (53.6 cm) long by 8.25 inches (20.9 cm) deep. The keyboard stands 3.5 inches (8.9 cm) high on four rubber feet.

3.2.1 Interface Cable

The Interface Cable is a 5-conductor, flexible cable with a shielded DIN plug which connects the PS 390 Keyboard to the front of the Peripheral Multiplexer. The cable may

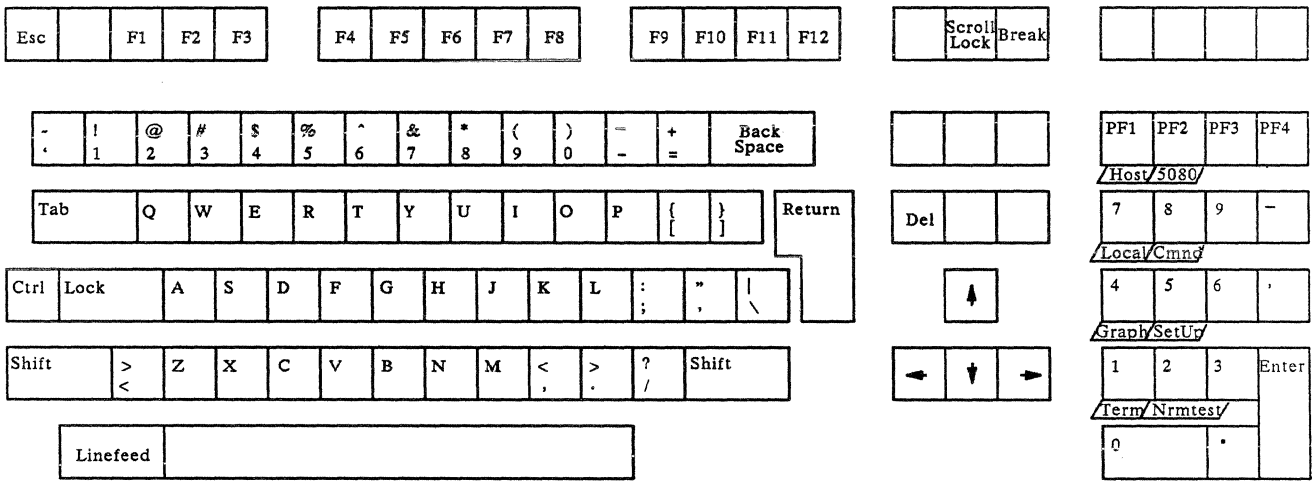


Figure 3.4: The PS 390 DEC VT-220 Style Keyboard

be stretched to permit many different work station arrangements.

3.2.2 Keyboard Operation

The PS 390 Keyboard allows the operator to input ASCII characters and other sequences to the Joint Control Processor by means of a typewriter-like keyboard. Keyboard operation is discussed in detail in the following paragraphs.

3.2.2.1 Data Entry

The PS 390 Keyboard features a layout that makes data entry fast and easy. All keys are momentary-closure devices.

The keys fall into eight general categories:

(a) Keyboard Function Control
(b) Alphabetic
(c) Standard Numeric
(d) Special Character
(e) Terminal Function
(f) PS 390 Function
(g) Numeric/Application Mode
(h) PS 390 Device Control

The Keyboard Function Control keys are used to modify the codes produced by other keys. In this way, characters are defined as uppercase, lowercase, control, etc. No codes are transmitted when these keys are depressed individually or in combination with each other.

The Alphabetic, Standard Numeric, Special Character, and Terminal Function keys all generate standard ASCII characters. Depressing any of these keys alone or in combination with SHIFT and/or CTRL causes 7-bit character codes or control codes to be transmitted from the keyboard to the Joint Control Processor.

The PS 390 Function, Numeric Keypad/Application Mode and Device Control keys are system-oriented. Depressing any of these keys alone or in combination with SHIFT and/or CTRL causes special two-byte PS 390 system sequences to be generated and transmitted to the JCP.

The following is a detail description with figures of the eight general key categories.

3.2.2.2 Keyboard Function Control Keys

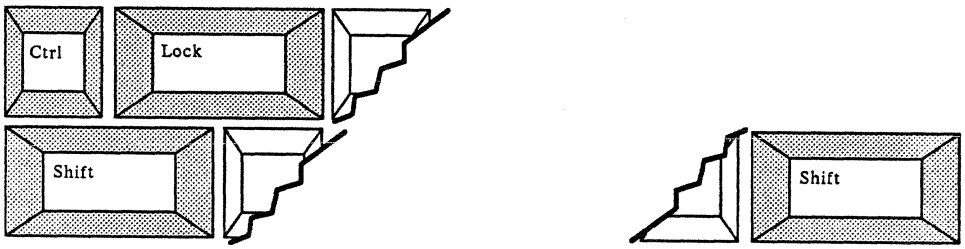


Figure 3.5: Keyboard Function Control Keys

The Keyboard Function Control keys (shown in gray in Figure 3.5) are un-encoded, local controls. No codes are transmitted when these keys are struck individually or in combination with each other. The Keyboard Function Control keys are used to modify the codes transmitted by other keys, as follows:

When either SHIFT key is depressed simultaneously with a displayable character key, the uppercase code for that key is generated. If the key does not have an uppercase function, the SHIFT key is ignored. For example, striking the **A** key causes the code B'01100001' for the character **a** to be transmitted; the sequence **SHIFT A** causes the code B'01000001 for the character **A** to be transmitted. Note that bit 6 is forced low to define an uppercase character.

When CTRL is depressed simultaneously with one of keys A-Z (uppercase only), the space bar, or the Special Character keys , [,],\, , or ?, an ASCII control code is generated. For example, the **CTRL Z** keyboard sequence causes the code B'00011010' to be generated. Note that the only difference between this code and that for **Z** (B'01011010') is that bit 7 is forced low to define the control code.

When the SHIFT and CTRL keys are depressed simultaneously, the CTRL function is selected in most cases. The only exceptions occur with the ~ and / keys. **SHIFT CTRL** ~ causes the control character **RS** (B'00011110') to be transmitted. **SHIFT CTRL** / causes the control character **US** (B'00011111') to be transmitted. The auto-repeat feature is enabled on all keys except: F1 → F12, SETUP, GRAPH, HOST, CMMD, LOCAL, TERM, LOCK, CTRL, SHIFT (both keys), RETURN, and all nu-

meric pad keys. When any other key is held down, repeated character transmission occurs. The rate is 15 ±2 Hz.

Depressing the SHIFT LOCK key enables the "shift lock" function. This is a shift operation that applies to all keys. Depressing either of the two shift keys causes the "shift lock" mode to be disabled.

3.2.2.3 Alphabetic Keys

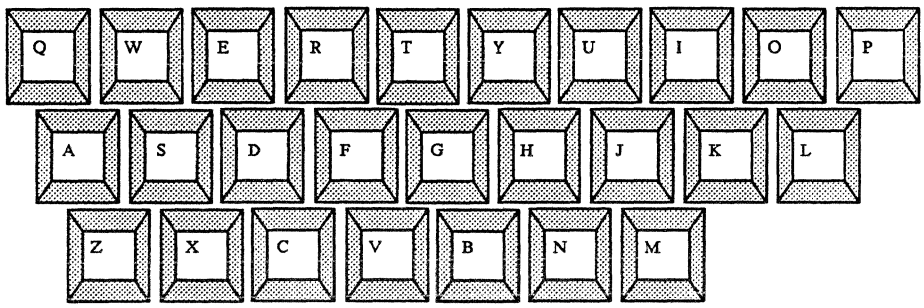


Figure 3.6: Keyboard Alphabetic Keys

The Alphabetic Keys (shown in gray in Figure 3.6) are used to produce uppercase and lowercase ASCII displayable character codes and ASCII control codes. Tables 3.5 through 3.6 show the code and character produced when each key is struck alone, with the SHIFT key, or with the CTRL key.

Table 3.5: Alphabetic Key Codes

Key	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
Label	*Code*	*Char*	*Code*	*Char*	*Code*	*Char*
A	X'61' 97	a	X'41' 65	A	X'01' 1	SOH
B	X'62' 98	b	X'42' 66	B	X'02' 2	STX
C	X'63' 99	c	X'43' 67	C	X'03' 3	ETX
D	X'64' 100	d	X'44' 68	D	X'04' 4	EOT
E	X'65' 101	e	X'45' 69	E	X'45' 5	ENQ
F	X'66' 102	f	X'46' 70	F	X'06' 6	ACK
G	X'67' 103	g	X'47' 71	G	X'07' 7	BEL
H	X'68' 104	h	X'48' 72	H	X'08' 8	BS
I	X'69' 105	i	X'49' 73	I	X'09' 9	HT
J	X'6A' 106	j	X'4A' 74	J	X'0A' 10	LF
K	X'6B' 107	k	X'4B' 75	K	X'0B' 11	VT
L	X'6C' 108	l	X'4C' 76	L	X'0C' 12	FF
M	X'6D' 109	m	X'4D' 77	M	X'0D' 13	CR
N	X'6E' 110	n	X'4E' 78	N	X'0E' 14	SO

Table 3.6: Alphabetic Key Codes - Continued

Key Label	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
	Code	*Char*	*Code*	*Char*	*Code*	*Char*
O	X'6F' 111	o	X'4F' 79	O	X'0F' 15	SI
P	X'70' 112	p	X'50' 80	P	X'10' 16	DLE
Q	X'71' 113	q	X'51' 81	Q	X'11' 17	DC1
R	X'72' 114	r	X'52' 82	R	X'12' 18	DC2
S	X'73' 115	s	X'53' 83	S	X'13' 19	DC3
T	X'74' 116	t	X'54' 84	T	X'14' 20	DC4
U	X'75' 117	u	X'55' 85	U	X'15' 21	NAK
V	X'76' 118	v	X'56' 86	V	X'16' 22	SYN
W	X'77' 119	w	X'57' 87	W	X'17' 23	ETB
X	X'78' 120	x	X'58' 88	X	X'18' 24	CAN
Y	X'79' 121	y	X'59' 89	Y	X'19' 25	EM
Z	X'7A' 122	z	X'5A' 90	Z	X'1A' 26	SUB

3.2.2.4 Standard Numeric Keys

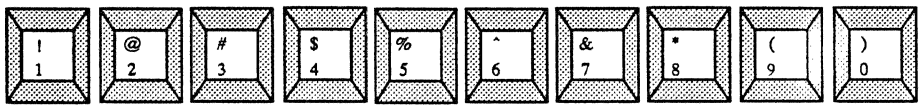


Figure 3.7: Keyboard Standard Numeric Keys

The shiftable Standard Numeric keys (shown in gray in Figure 3.7) are similar to the shiftable numeric/symbol keys that appear on a typewriter; they generate ASCII displayable numbers and symbols. The CTRL key is ignored when used with these keys. Table 3.7 shows the code and character produced when each key is struck alone, with the SHIFT key, or with the CTRL key.

Table 3.7: Standard Numeric Key Codes

Key Label	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
	Code	*Char*	*Code*	*Char*	*Code*	*Char*
0	X'30' 48	0	X'29' 41	)	X'30' 48	0
1	X'31' 49	1	X'21' 33	!	X'31' 49	1
2	X'32' 50	2	X'40' 64	@	X'32' 50	2
3	X'33' 51	3	X'23' 35	#	X'33' 51	3
4	X'34' 52	4	X'24' 36	$	X'34' 52	4
5	X'35' 53	5	X'25' 37	%	X'35' 53	5
6	X'36' 54	6	X'5E' 94	^	X'36' 54	6
7	X'37' 55	7	X'26' 38	&	X'37' 55	7
8	X'38' 56	8	X'2A' 42	*	X'38' 56	8
9	X'39' 57	9	X'28' 40	(	X'39' 57	9

3.2.2.5 Special Character Keys

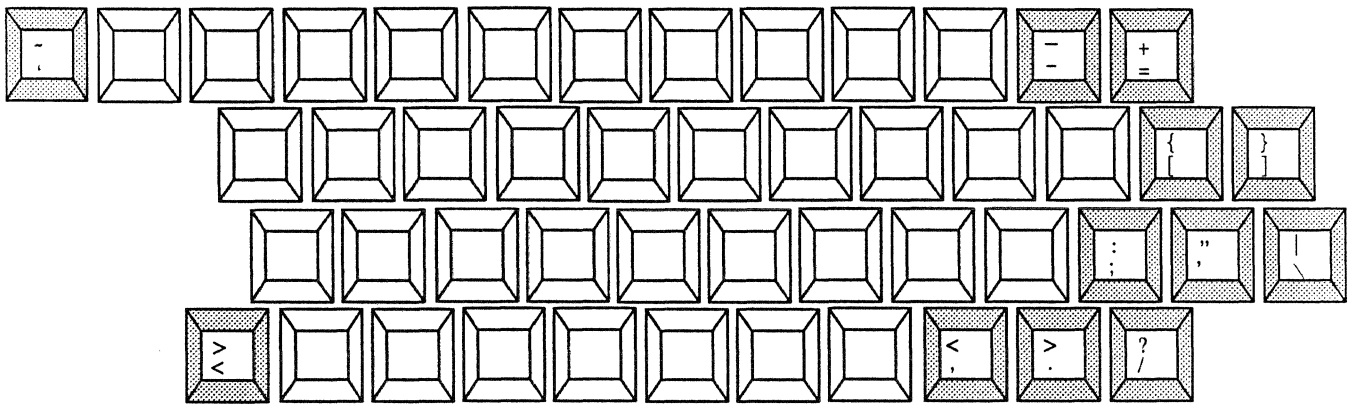


Figure 3.8: Keyboard Special Character Keys

The shiftable Special Character keys (shown in gray in Figure 3.8) are used to produce both ASCII displayable characters and ASCII control characters. Table 3.8 shows the codes and characters produced when these keys are activated alone, with the SHIFT key, and with the CTRL key. Note the varying response given to the CTRL key; in some instances, the unshifted key character is produced. In other cases, a control character is generated.

Table 3.8: Special Character Key Codes

Key	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
Label	*Code*	*Char*	*Code*	*Char*	*Code*	*Char*
_	X'2D'	-	X'5F'	_	X'2D'	-
-	45	(minus)	95	(underline)	45	(minus)
+	X'3D'		X'2B'		X'3D'	
=	61	=	43	+	61	=
~	X'60'		X'7E'		X'1E'	
`	96	~	126	`	30	RS
{	X'5B'		X'7B'		X'1B'	
[	91	[	123	{	27	ESC
}	X'5D'		X'7D'		X'1D'	
]	93	]	125	}	29	GS
|	X'5C'		X'7C'		X'1C'	
\	92	\	124	|	28	FS
:	X'3B'		X'3A'		X'3B'	
;	59	;	58	:	59	;
"	X'27'		X'22'		X'27'	
'	39	'	34	"	39	'
<	X'2C'		X'3C'		X'2C'	
,	44	,	60	<	44	,
>	X'2E'		X'3E'		X'2E'	
.	46	.	62	>	46	.
?	X'2F'		X'3F'		X'1F'	
/	47	/	63	?	31	US
>	X'2C'		X'2E'		X'2C'	
<	44	<	46	>	44	<

3.2.2.6 Terminal Function Keys

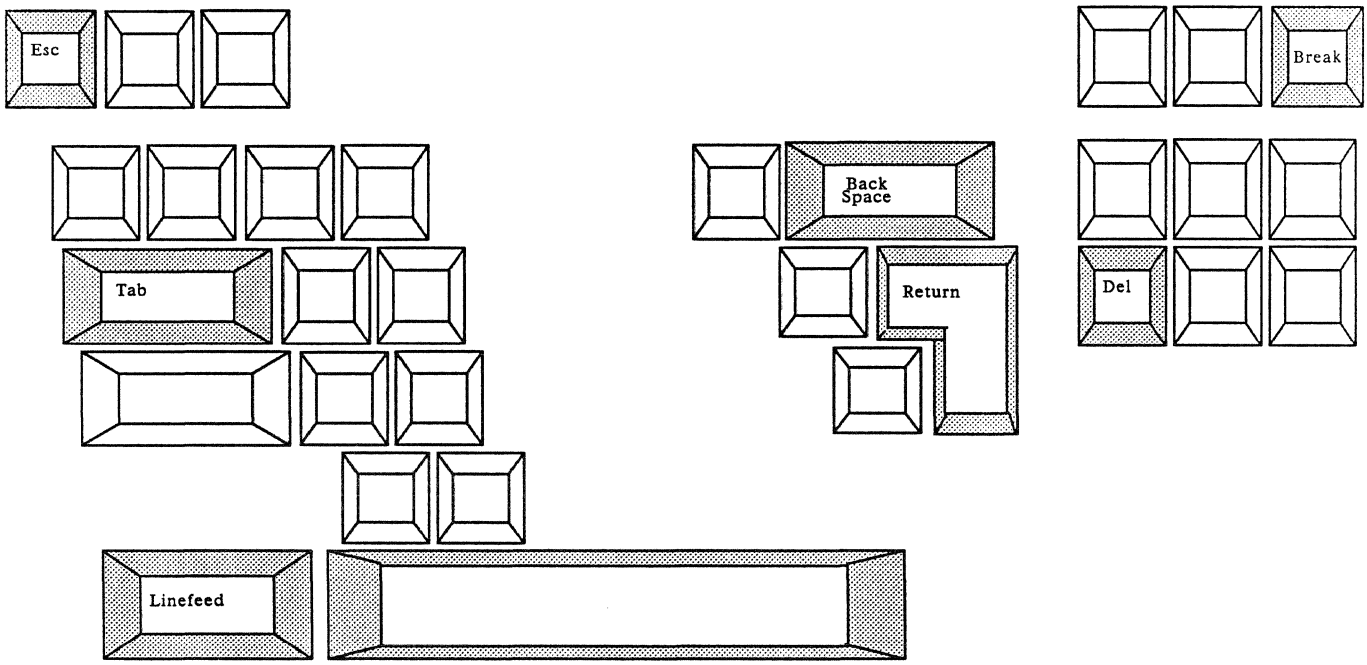


Figure 3.9: Keyboard Terminal Function Keys

The Terminal Function keys (shown in gray in Figure 3.9) produce codes used by a typical video display terminal. These keys enable an operator to generate any commonly used terminal control character with a single keystroke. (The codes produced by these keys are identical to those generated by the conventional two-key control sequences described in Table 3.9.)

Note that the SHIFT and CTRL keys have no effect on the codes produced by the Terminal Function keys, except for the **CTRL Space Bar** sequence that generates an

ASCII **NUL** character.

Table 3.9 lists the codes and characters generated by the Terminal Function keys.

Table 3.9: Terminal Function Key Codes

Key	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
Label	*Code*	*Char*	*Code*	*Char*	*Code*	*Char*
BREAK	X'A0' 160		X'A0' 160		X'A0' 160	
BACK SPACE	X'08' 8	BS	X'08' 8	BS	X'08' 8	BS
DEL	X'7F' 127	DEL	X'7F' 127	DEL	X'7F' 127	DEL
RETURN	X'0D' 13	CR	X'0D' 13	CR	X'0D' 13	CR
LINE FEED	X'0A' 10	LF	X'0A' 10	LF	X'0A' 10	LF
ESC	X'1B' 27	ESC	X'1B' 27	ESC	X'1B' 27	ESC
TAB	X'09' 9	HT	X'09' 9	HT	X'09' 9	HT
(none; space bar)	X'20' 32	(space)	X'20' 32	(space)	X'00' 0	NUL

3.2.2.7 PS 390 Function Keys

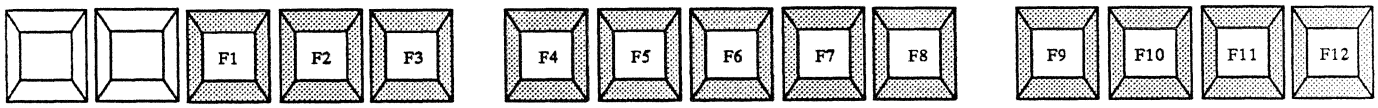


Figure 3.10: Keyboard PS 390 Function Keys

The PS 390 Function keys (shown in gray in Figure 3.10) are used to transmit special 2-byte system sequences.

Table 3.10: PS 390 Function Key Codes

Key Label	*Key Alone Code*	*Key+SHIFT Code*	*Key+CTRL Code*
F1	X'1661	X'1641'	X'1601'
F2	X'1662	X'1642'	X'1602'
F3	X'1663'	X'1643'	X'1603'
F4	X'1664'	X'1644'	X'1604'
F5	X'1665'	X'1645'	X'1605'
F6	X'1666'	X'1646'	X'1606'
F7	X'1667'	X'1647'	X'1607'
F8	X'1668'	X'1648'	X'1608'
F9	X'1669'	X'1649'	X'1609'
F10	X'166A'	X'164A'	X'160A'
F11	X'166B'	X'164B'	X'160B'
F12	X'166C'	X'164C'	X'160C'

3.2.2.8 Numeric/Application Mode Keys

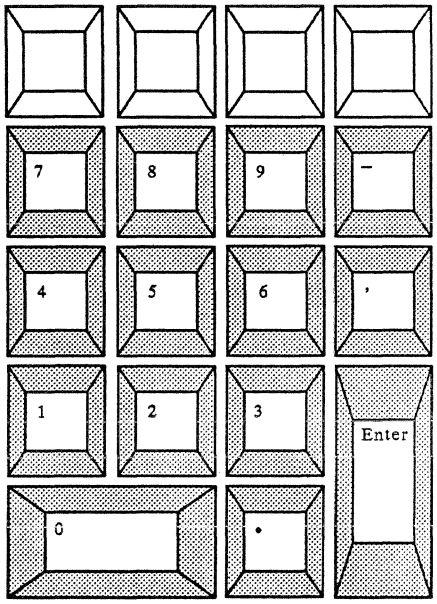


Figure 3.11: Keyboard Numeric/Application Mode Keys

The Numeric/Application Mode keys (shown in gray in Figure 3.11) generate special 2-byte PS 390 system sequences similar to those produced by the PS 390 Function keys.

Note that neither SHIFT nor CTRL affects the ENTER key, and that no codes are modified by the CTRL key.

Any code generated by a Numeric/Application Mode key may be duplicated by entering CTRL SHIFT V, followed by the appropriate displayable character or control character.

Table 3.11 illustrates the codes and characters produced by the Numeric/Application Mode keys.

Table 3.11: Numeric/Application Mode Key Codes

Key Label	*Key Alone*		*Key+SHIFT*		*Key+CTRL*	
	Code	*Char*	*Code*	*Char*	*Code*	*Char*
0	X'1630'		X'1629'		X'1630'	
1	X'1631'		X'1621'		X'1631'	
2	X'1632'		X'1640'		X'1632'	
3	X'1633'		X'1623'		X'1633'	
4	X'1634'		X'1624'		X'1634'	
5	X'1635'		X'1625'		X'1635'	
6	X'1636'		X'165E'		X'1636'	
7	X'1637'		X'1626'		X'1637'	
8	X'1638'		X'162A'		X'1638'	
9	X'1639'		X'1628'		X'1639'	
.	X'162E'	.	X'163E'	>	X'162E'	.
,	X'162C'	,	X'163C'	<	X'162C'	,
-	X'162D'	(minus) -	X'165F'	(underline) _	X'162D'	-
ENTER	X'160D'	CR	X'160D'	CR	X'160D'	CR

3.2.2.9 PS 390 Device Control Keys

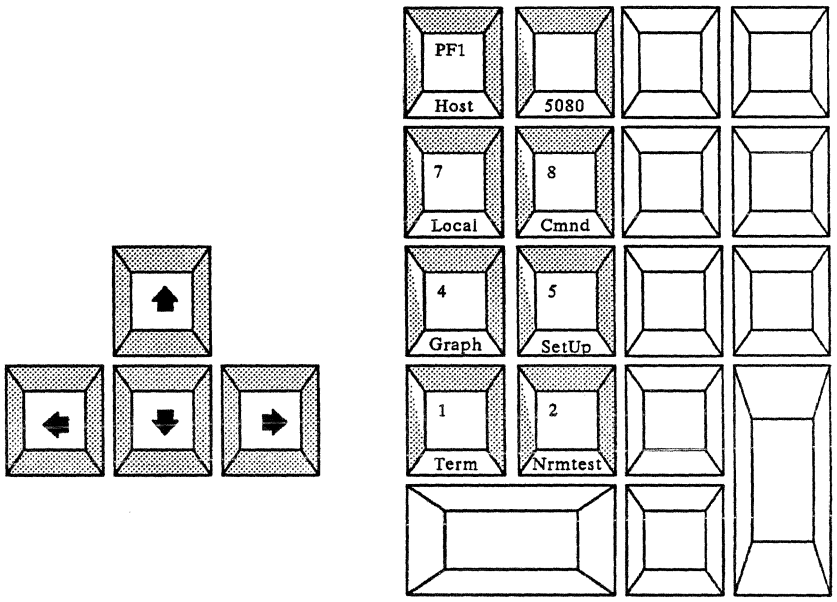


Figure 3.12: Keyboard PS 390 Device Control Keys

The Device Control keys (shown in gray in Figure 3.12) generate two-byte sequences similar to those described in Sections 3.2.2.7 and 3.2.2.8. The codes produced by these keys are modified by SHIFT and CTRL as shown in Table 3.12.

Any code generated by a Device Control key may also be produced by entering CTRL SHIFT V, followed by the appropriate displayable character or control character.

Table 3.12: PS 390 Device Control Key Codes

Key Label	*Key Alone Code*	*Key+SHIFT Code*	*Key+CTRL Code*
1 TERM	X'1673'	X'1653'	X'1613'
2 NRMTST	X'1632'	X'1640'	X'1644'
4 GRAPH	X'1634'	X'1624'	X'1670'
5 SET UP	X'1635'	X'1625'	X'166F'
7 LOCAL	X'1637'	X'1626'	X'1652'
8 CMND	X'1638'	X'162A'	X'1612'
←	X'1677'	X'1657'	X'1617'
→	X'1678'	X'1658'	X'1618'
↑	X'1679'	X'1659'	X'1619'
↓	X'167A'	X'165A'	X'161A'
PF1 HOST	X'A9'	X'A9'	X'1672'
PF2 5080	X'AA'	X'AA'	X'AA'
PF3	X'AB'	X'AB'	X'AB'
PF4	X'AC'	X'AC'	X'AC'

The Cursor Up key becomes Scroll Up when shifted.
The Cursor Down key becomes Scroll Down when shifted.

3.2.3 Communications Interface

The keyboard communicates with the JCP through the Peripheral Multiplexer using a RS-232C line receiver and line driver. The keyboard operates at 1200 baud.

3.2.4 Dual Function Switchable Keyboard

Unique features and IBM key codes for the Dual Function Switchable keyboard will be described as they become available.

3.3 The 32 Key Lighted Function Buttons

The 32-Key Lighted Function Buttons (hereafter called the Buttons) consists of an array of 32 lighted function keys arranged in a 6×6 matrix without the key at each of the four corners being present. The Joint Control Processor sends the message to the Buttons box that lights the keys which are candidates to be selected to invoke specific program functions. The same message also turns off some of the lights which are already on. This cues the operator of the station to know that he may select one of the lighted keys by depressing the key. Upon depression, the Buttons box sends a message to the Joint Control Processor which indicates that a specific key has been depressed. The software can then take action(s) based upon the key selection.

3.3.1 Light Control

For the purpose of turning the lights of the Buttons box on or off, the lights are logically grouped into eight groups of four lights each. The lights of the box are then turned on and of respectively by sending a message consisting of one to eight bytes to it. The four more-significant bits of each byte contains the identification number for a four-light group; the four less-significant bits contain a mask which turn on (if the corresponding bit is set) or off (if the bit is clear) the light. This is shown in Figure 3.13 where the Group Number is binary 0000 thru 0111 and Light Mask 1's and 0's turn lights on and off.

The Function Button Light Groups are defined in Table 3.13.

Any byte or combination of bytes may be sent in a message, depending on which of the lights must be turned on or turned off. Turning all lights on, turning all lights off or changing the state of at least one byte of each of the eight groups would require an eight-byte message to be sent. Changing the state of one to four lights in a single four-light

Figure 3.13: Function Button Light Control Message Byte

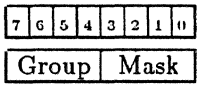


Table 3.13: Function Button Light Groups

Group Number	*Description*
b'0000'	Group for lights 1 → 4
b'0001'	Group for lights 5 → 8
b'0010'	Group for lights 9 → 12
b'0011'	Group for lights 13 → 16
b'0100'	Group for lights 17 → 20
b'0101'	Group for lights 21 → 24
b'0110'	Group for lights 25 → 28
b'0111'	Group for lights 29 → 32

group would require only a one-byte message to be sent.

3.3.2 Reporting Selections

The Buttons box reports that a key has been depressed simply by sending a single byte to the Joint Control Processor. The value of the byte is given by adding the hexadecimal value of the key number to the hexadecimal value x'3F'. Thus the first sixteen keys are numbered x'40' to x'4F' and the second group of sixteen keys are numbered x'50' to x'5F'. Only one key depression per message is reported.

3.3.3 Self-Test Command and Report

The Buttons box includes a self-test command and report that is used for diagnostics and optionally for initialization confidence tests. The command is a single byte: x'80'. The response is a four-byte sequence as shown in Table 3.14.

Table 3.14: Function Box Self Test Responses

Byte 1	64H, Hardware ID for the Button Box.
Byte 2	xxH, where xx is the firmware revision level. This should begin with 01H.
Byte 3	00H if ROM and RAM test successful and 3EH if ROM or RAM test failed, (RAM and ROM refer to processor chip), or 3DH if key down on Self Test (3E supersedes 3D)
Byte 4	00H on successful test, or xxH, where xx is code of keydown at Self Test.

3.3.4 Transmission Characteristics

The data sent to and from the Buttons box is asynchronous data with each byte containing eight data bits without parity plus one start bit and one stop bit. The data transmission rate of the Buttons box is 9600 baud.

3.4 The Control Dials

3.4.1 Functional Characteristics

The Control Dials (hereafter called the Dials) consists of an array of 8 shaft encoders arranged in a 2 column x 4 row design with the number 1 dial being the upper left-hand dial and the number 5 dial being the upper right-hand dial when the Dials are situated in the vertical orientation. When the Dials are situated in the horizontal orientation, the number 1 dial is the lower left-hand dial and the number 5 dial is the upper left-hand dial. The Dials report to the Joint Control Processor the number of counts rotated between sampling intervals. The Joint Control Processor may specify the number of counts to be accumulated between sampling intervals and may set a sampling time for all the dials.

3.4.1.1 Dial Responses to the Host

The Dials output relative delta values only; i.e., each dial's position is reported in terms of its last sample location. The data format used to report the count is:

Byte Number	*Description*
1	Control V = '00010110'
2	Byte = '00000nnn', Where nnn is a binary number 000 thru 111 (0 thru 7 decimal which specifies the dial.)
3	Most significant byte of a 16-bit signed integer (sign indicates direction).
4	Least significant byte of the 16-bit signed integer (two's complement notation).

3.4.1.2 Commands to the Dials from the Host

There are two commands to which the Dials box must respond. The first is in the same format as the response message except that the second byte is '100xxnnn' and no sign is legal on the 16-bit integer. It specifies the delta value which must be accumulated before the delta count is reported to the host, i.e., how many counts between reports.

The second command is formatted as follows and applies a sampling time to all the dials:

Byte Number	*Description*
1	Control V = '00010110'
2	Control Byte = '1x1xxxxx', (x=don't care)
3	Reserved unused byte.
4	Time count in binary, Where x'05' = 60 samples/second Where x'0A' = 30 samples/second Where x'1E' = 10 samples/second

This time indicates how often the dials box wakes up to see if sufficient counts have been accumulated on any dial to respond to the processor.

3.4.2 Transmission Characteristics

The data sent to and from the Dials box is asynchronous data with each byte containing eight data bits without parity plus one start bit and one stop bit. The data transmission rate of the Dials box will be 9600 baud.

3.5 The Data Tablet

There are two data tablets available for use with the Low Cost Peripheral Set. They are the 6" × 6" and the 12" × 12" tablets with a four-button cursors. Both are alike except for their active areas and both provide digitizing and picking functions for the PS 390.

The tablets use +12 VDC @ 300 mA max. The power is provided by the multiplexer on the 7-pin Micro DIN connector provided.

The Data Tablets transform graphic information into digital data suitable for transmission to the Joint Control Processor (JCP). The data tablets use a stylus or a four button cursor to identify coordinates. Touching the pen-like stylus to any position on the data tablet transforms the coordinates of that position into their digital equivalents. The cursor contains a crosshair sight that permits the user to enter data with precise accuracy.

3.5.1 Data Tablet Microprocessor

The data tablet's microprocessor is an 8-bit Intel 8035. The microprocessor outputs two control signals that are used to gate the X pulse and the Y pulse to the data tablet. The microprocessor also controls communications with the host processor that is located on the JCP card.

If the microprocessor fails, the data tablet will not respond to the TABD1A diagnostic. In this case, the microprocessor should be replaced.

3.5.2 Operating Modes

Data tablet modes and the sampling rates may be controlled externally under program control or internally by switches on the logic card. The positions of the internal switch determine the power-up mode and sampling rate. The following operating modes are available:

- Point Mode
  - Pressing the stylus on the tablet or pressing a button on the cursor outputs one X, Y coordinate pair (sample) in the appropriate format.
- Stream Mode
  - X, Y coordinate pairs (samples) are generated continuously at the selected sampling rate when the stylus or cursor is near the active area of the tablet. Pressing the stylus to the tablet or depressing a button on the cursor puts the flag character (F) bit in the output string.

- Switched Stream Mode
  - Pressing the stylus or a button on the cursor continuously outputs X, Y coordinate pairs at the selected sampling rate until the stylus is lifted or the button is released.

The data tablet has a six-position switch that sets the mode of operation and the rate at which the coordinate data are output to the processor. The Mode and Rate Controls on the data tablet are mounted on SW 2. Positions 1 and 2 are mode switches and Positions 3, 4, and 5 are rate switches. Switch 6 is not used. The system reset switch is mounted externally at the rear of the lower frame.

Both the mode and the sampling rate may be changed under program control from the PS 390 by sending the data tablet an ASCII character.

3.5.3 Power Requirements

The data tablet is shipped with a connector that mates with the power input connector located at the rear of the data tablet. The pin assignments that apply to this connector are shown in Table 3.15.

Table 3.15: Data Tablet Pin Assignments

Pin #	Function
1	Ground
2	Transmit data (From Device)
3	Receive data (To Device)
4	-12 VDC
5	+ 5 VDC
6	+12 VDC
7	Device Present (Connected to Pin 1)
Shell	Protective Ground (ESD Shield)

3.5.4 Data Tablet/PS 390 interface

The data tablet communicates with the PS 390 via an RS-232 asynchronous cable. Each character is transmitted as a complete self-contained message consisting of an ASCII data character with even or odd parity (POE) preceded by a start bit and followed by

one or two stop bits, depending on the strap option selected (HCB). The bit polarity of the transmitted data is low level mark, high level space in the following format:

Start Bit	*Seven Data Bits*	*Parity*	*Stop*	Stop

3.5.4.1 Binary Data Format (Switch 1, Position 7 ON)

The binary formatted RS-232 interface is a five byte count output. Binary format is as follows:

Binary Format								
Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1	P	1	F3	F2	F1	F0	0	0
2	P	0	X5	X4	X3	X2	X1	X0
3	P	0	X11	X10	X9	X8	X7	X6
4	P	0	Y5	Y4	Y3	Y2	Y1	Y0
5	P	0	Y11	Y10	Y9	Y8	Y7	Y6

3.5.4.2 Remote Control via RS-232

The data tablet is a stand-alone microprocessor-driven device that can be remotely programmed. The Joint Control Processor controls remote operation of the data tablet. The following conditions must exist for remote control of the data tablet:

- All internal mode and rate controls (SW 2) must be inactive or in the OFF condition.
- Data going to the data tablet must be at the same baud rate as the data transmitted from the data tablet.
- Data tablet command data must be input on J1 Pin 3 with a bit polarity of low level mark, high level space.
- One of the binary data transmission codes shown in Table 3.16 must be selected.

Table 3.16: Binary Data Transmission Codes

Mode	*Binary Rate*	*Uppercase* ASCII Character
Stop	-	S
Point	-	P
Switched Stream	2	@
	4	A
	10	B
	20	C
	35	D
	70	E
Stream	141	F
	141	G
	2	H
	4	I
	10	J
	20	K
	35	L
	70	M
	141	N
	141	

Note: Rate is calculated as coordinate pairs per second at 19,200 baud. All other rates are dependent on baud rates.

3.5.4.3 RS-232 Unit Switch Settings & Strap Options for 600 Series PROMs

Data tablets are shipped with a standard setting from the factory. The following sections describe strap and switch settings for non-standard strap and switch settings. Refer to the GTCO Users Manual, for all switch and strap locations.

Switch 1 (Format/Calibration) This nine-position switch controls the output data format as follows:

Table 3.17: RS-232 Switch Settings

Position	*Effect*
1	Do not adjust–Factory Set
2	Do not adjust–Factory Set
3	Do not adjust–Factory Set
4	Do not adjust–Factory Set
5	Do not adjust–Factory Set
6	Not Used
7	ON–Serial Binary Output (No CRLF transmitted when in Serial Binary (Position 8) *OFF-ASCII BCD Output
8	*ON-Carriage Return Line Feed (CRLF). This adds line feed to the end of output data format. OFF-Carriage Return (CR) only.
9	*ON–English (0.005" Resolution) OFF–Metric (0.1 mm Resolution)

* Factory settings.

Note

Switches 1→5 are factory set calibration switches. They should not be changed unless the tablet portion of the device is changed. Refer to the GTCO User's Manual.

Switch 2 (Mode/Rate) This six-position switch controls the sampling mode (Point, Switch Stream, or Continuous Stream) and the sampling rate (X-Y coordinate pairs per second). The switch is factory set in the Continuous Stream Mode at 200 samples per second. To operate under program control, set all internal position switches to OFF.

The PS 390 Data Tablet operates at 9600 baud, sending 105 serial binary samples per second. Due to the limitations of serial baud rate transmit time, the maximum sampling rate is automatically limited to the sampling rates shown in Table 3.18.

Table 3.18: Data Tablet Sampling Rates

Baud Rate	*Serial ASCII BCD*	*Serial Binary*
Maximum Sampling Rate		
28800	85	166
19200	68	141
9600	46	105
4800	28	65
2400	16	37
1200	9	20
300	2	5

Switch 7 and Pluggable Program Strap BA (Baud Rate) Both Switch 7 and the Pluggable Strap BA must be set to select the desired baud rate. One of the ten positions on Switch 7 must be set to ON and the blue pluggable strap must be over the center pin and the A pin (or over the center pin and the B pin). Only one position on Switch 7 may be on at a time. The baud rate is factory set with Position 2 ON on Switch 7 and pluggable strap BA over Pin B and the center pin. Table 3.19 shows the baud rates that may be selected.

POE Strap (Parity) Polarity can be odd or even and is controlled by a wire jumper soldered into the two points on the circuit card labeled POE. The RS-232 Data Tablet is shipped with no strap and in the even parity mode.

HCB Strap (Stop Bits) There may be one or two stop bits transmitted. The number of stop bits transmitted is controlled by a wire jumper soldered into the two points on

Table 3.19: Baud Rate Selection

Switch 7 (Position ON)	*Blue Pluggable Strap BA*	
	Strap A	*Strap B*
1	19200	19200
2*	28880	9600*
3	14400	4800
4	7200	2400
5	3600	1200
6	1800	600
7	900	300
8	450	150
9	225	75
10	112.5	–

* Factory setting.

the card labeled HCB. The RS-232 unit is shipped with a strap and transmits one bit. The PS 390 TABD1B diagnostic tests the data tablet.

3.6 The Optical Mouse

The Optical Mouse transforms position information into a digital form acceptable to the Joint Control Processor (JCP). The optical mouse uses a three-button mouse unit in conjunction with a reflective pad to provide x and y-axis position information. The cursor (an X) moves around on the screen in response to movement of the mouse across the pad.

The mouse uses red and infrared LED's reflecting off the pad to provide directional information to the control logic in the mouse. This movement is then translated into absolute x and y position information similar to that provided by a PS 390 Data Tablet. The data is transmitted serially to the PS 390 through the peripheral multiplexer.

3.6.1 Operating Modes

The mouse operates in what is called MM Series Delta Data Protocol. This is a 3 byte format that defines change in mouse position as delta movement in X and Y. The mouse also operates in Exponential Scaling Mode. This means that the faster the mouse is moved across the pad, the larger the data increments will be. This allows a single move across the pad to produce a complete movement of the cursor across the screen.

3.6.2 PS 390 Runtime Operation

The PS 390 contains one system function to interface the mouse to an application. It is called MOUSEIN, and is an instance of f:mouse. The Mouse is connected to Mouse Port on the Multiplexer, or to Port D on the Data Concentrator. The MOUSEIN function is already connected to the pick_location just the same as TABLETIN.

The MOUSEIN function instance has the same outputs as TABLETIN. There are 4 inputs to the Mouse function.

(a) String – Data from the Mouse
(b) Integer – Counts full scale
(c) String – Output queues enable/disable message
(d) Vec2d – New cursor position

Input 1 is a string of data from the Mouse in the format shown in Table 3.20. Input 2 is an integer specifying the number of counts to map to a cursor movement across the screen. The default is 2200. Input 3 is a string of up to eight characters (characters in strings longer than eight are ignored) consisting of either T or F. This is a positional indication of the enable or disable of a particular output. For example, the string 'TTTFFT' would enable outputs 1, 2, 3, and 6; the string 'TTFFFFF' would enable output from 1 and 2 only.

Note: Only the F is checked for. Therefore 'XXFFyyy' would be the same as 'TTFFTTT'. The default is 'TTTTTTTT'.

Input 4 is a 2-D vector that will position the cursor on the screen at the point specified. the value should be in the range of -1.0 to 1.0. The default is 0.0.0.0.

The runtime will support the use of both the tablet and the mouse on the same system. A special "Y" power connector will be required if both are to be used on a data concentrator since there is only one 9-pin D power connection.

3.6.3 Mouse/PS 300 Interface

There are two versions of the mouse. One is for use with the Data Concentrator and the other is for use with the Peripheral Multiplexer. The pin assignments on the data concentrator version are identical to the data tablet. The pinouts for the MUX version are as follows:

Pin #	Function
1	Ground
2	Transmit data (From Mouse)
3	Receive data (To Mouse)
4	-12 VDC
5	Not Used
6	+12 VDC
7	Device Present (Connected to Pin 1)
Shell	Protective Ground (ESD Shield)

With the connector of the mouse cable facing you the pins are numbered according to the following diagram:(* = connector key)

3.6.3.1 Baud Rate

The mouse is configured to run at 9600 Baud over the serial asynchronous interface.

3.6.3.2 Data Format

The data format consists of 3 bytes which are assigned as follows:

	MSB 7	6	5	4	3	2	1	LSB 0
Byte 1	1	0	0	Sign X	Sign Y	Left	Center	Right
Byte 2	0	X_6	X_5	X_4	X_3	X_2	X_1	X_0
Byte 3	0	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0

Table 3.20: Mouse Data Format

The bit positions Left, Center, and Right indicate the status of the three buttons on the Mouse. A one in the positions indicates that the button is down.

Part II

Customer Engineering

Chapter 4

The PS 390 Diagnostics

The main hardware components of the Raster Backend portion of the PS 390 which must be tested with the diagnostic software are:

(a) The Input FIFO
(b) The Master Bit Slice Processor
(c) The Endpoint Pipeline
(d) The Frame Buffer
(e) The Video Output System
(f) The New Peripherals

The diagnostic routines for the PS 390 Raster Backend and the new peripheral set are:

- RBED0A
  - Bitslice Processor Diagnostic
- RBED1A
  - Endpoint Pipeline Diagnostic
- RBED2A
  - Pixel Processor, Frame Buffer and Video Card Diagnostic
- MPLSD0B
  - PLS Card and Input FIFO Diagnostic
- KBDD0C
  - New Keyboard Diagnostic
- BTND0B
  - Function Buttons Diagnostic (Old/New)

- CDLD0B
  - Control Dials Diagnostic (Old/New)
- MSED0B
  - Optical Mouse Diagnostic
- TABD1B
  - GTCO Digitizer Tablet Diagnostic

4.1 Bitslice Processor Diagnostics - RBED0A

4.1.1 Hardware Overview

The PS 390's Master Bitslice Processor obtains endpoints and commands from the Input FIFO. Endpoints are formatted into a packet of data and then sent to the Endpoint Pipeline. Commands received by the Master Bitslice are decoded and then executed by the Master Bitslice.

There are four busses on the Bitslice card. The 29117 processor can latch data in only from the D-Bus. It can provide latch data out via the Ybus. Communication between the Y and D bus is possible. The branch bus communicates between the Vector ram and the 29110 Microsequencer. The Immediate bus is a multi-purpose bus. The Bitslice Processor has the following components:

(a) Common Bus Maintenance Register.
(b) Writeable Control Store.
(c) Execution Register.
(d) 29110 Microsequencer.
(e) 29117 Microprocessor.
(f) Immediate Field Register.
(g) Scratch RAM.
(h) Function Lookup Table EPROM.
(i) Vector RAM and Branch Bus.
(j) AMD 29517A Multiplier.
(k) Bus Decoders.
(l) Y and D Bus Transceivers.
(m) Common Bus Interrupt Generator.
(n) Common Bus DMA Interface.

4.1.2 Testing Strategy

The strategy for testing the Bitslice processor is to first test the Maintenance Register on the Common Bus. Then the Writeable Control Store and Execution Register are tested. The 29110 microsequencer and the Condition Code Multiplexer are then tested, followed by tests for the 29117 Microprocessor. Finally the Immediate Field Register, Scratch RAM, Lookup Table, Vector RAM, Multiplier, and Y to D Bus Latches are tested.

Phase 1 - Common Bus Maintenance Register
Phase 2 - Execution Register and Diagnostic Shift Loop
Phase 3 - Y to D Bus, Immediate Field Register
Phase 4 - Writeable Control Store
Phase 5 - 29117 Internal ALU Registers (32)
Phase 6 - Interrupt Testing
Phase 7 - 29110 Microsequencer and Condition Code Multiplexer
Phase 8 - 29117 Microprocessor Instruction Confidence
Phase 9 - Scratch RAM
Phase 10 - Vector RAM
Phase 11 - Function Lookup Table
Phase 12 - AMD 29517A Multiplier
Phase 13 - Common Bus Direct Memory Access (DMA)

4.1.3 Description of Tests

4.1.3.1 Phase 1 - Common Bus Maintenance Register

Phase 1 tests out the Maintenance Register located at X'FFF030'. Phase 1 has 2 subphases.

This phase tries to read and write to the Maintenance Register. If a bus error occurs, a diagnostic interrupt handling routine detects it and an error message is reported. The different bits of the Maintenance Register are then tested by toggling each bit high then low.

Subphase 1 of Phase 1 attempts to read the maintenance register. If a bus error happens during the reading of the maintenance register the following error occurs.

```
Common bus read error for Raster Back End MR: (PS 390
        Maintenance register's contents).
```

Subphase 2 of Phase 1 attempts to write to the maintenance register and to set and reset various bits of the maintenance register.

If a bus error occurs during the writing of the maintenance register the following error occurs.

```
Common bus write error for Raster Back End MR: ( PS 390
        Maintenance register's contents ).
```

The following bits in the maintenance register are tested by setting and resetting them – bits 0, 1, 2, 3, 4, 5, 13, 14.

If an error occurs while setting a bit the following error occurs.

```
Error in setting a RBE Maintenance Register bit
  Expected: X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

If an error occurs while resetting a bit the following error is displayed.

```
Error in resetting a RBE Maintenance Register bit.
  Expected: X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

4.1.3.2 Phase 2 - Execution Register

The Execution Register is checked to see if it can be written and read to correctly.

Phase 2 checks the data paths on the shadow register, the execution register and the immediate field. This phase shifts patterns into and out of these registers a bit at a time through the maintenance register. Phase 2 has 3 subphases.

Subphase 1 tests the shadow register. The shadow register is 80 bits long. This register is tested with the standard set of diagnostic patterns:

If the 80 bit pattern sent is different than the 80 bit pattern received the following error message is displayed.

```
Data mismatch in 29818 shadow reg while testing (bits <79 - 0>)
Expected: X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

Subphase 2 tests the Execution Register. This register is 80 bits long, but the last 16 bits are the Immediate Field, so only 64 bits are tested in this subphase. Data is shifted into the Shadow Register, clocked to the Pipe Register, then they are clocked back to the Shadow Register and shifted out.

If the 64 bit pattern sent is different than the 64 bit pattern received the following error message is displayed.

```
Data mismatch in 29818 execution reg while testing (bits <79 - 0>)
Expected: X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

Subphase 3 tests the 16 bit immediate field with a 16 bit diagnostic pattern. If the 16 bit pattern sent is different then the 16 bit pattern received the following error message is displayed.

```
Error found in the Immediate field.
  Expected: X'xxxx' Received  X'yyyy' Bits in Error: X'zzzz'
```

If the bit 16 of the microword (*HOLDWIM) cannot be set or if the Flip-flop this signal drives is not working, errors in the immediate field readback will occur.

4.1.3.3 Phase 3 - Y to D Bus Test, Immediate Field Register

Phase 3 tests the D to Y bus path and in doing so also tests the 16-Bit Immediate Register.

These Bus Transceivers are accessed via the Bitslice Processor. They are first tested to see if they can move data from the Y to the D Bus as well as from the D to the Y Bus, and then a data lines test is performed on each latch. This test is single stepped. A microcoded full speed test of this transceiver is performed in one of the later phases.

Subphase 1 tests out the D to Y bus path and the Immediate Register with the 16 bit diagnostic pattern set.

If the 16 bit pattern sent is different then the 16 bit pattern received the following error message is displayed:

```
Data mismatch while testing D to Y Bus communication.
  Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

4.1.3.4 Phase 4 - Writeable Control Store

The Writeable Control Store (WCS) for the PS 390 is 80-bits by 4096 words. Access to the Control Store is via the Common Bus Interface. The Control Store test increments the program counter from 0 to 2047 and reads, writes, and checks test patterns at each location.

The control store has 4096 words. Each word is 80 bits wide. Phase 4 has 3 subphases.

Subphase 1 tests out the data lines to the writeable control store. A 16 bit diagnostic pattern set is clocked into a location in the WCS and read back. This pattern is repeated

to fill the entire 80 bit microword. If the 80 bit pattern sent is different then the 80 bit pattern received the following error message is displayed.

```
Datalines error writing to Control store (bits <79 - 0>)
 Expected: X'xxxx' Received  X'yyyy' Bits in Error: X'zzzz'
```

Subphase 2 tests out the 12 address lines to the writeable control store. First each address line is tested to see if is stuck high or low, then each possible pair of address lines are tested to see if they are stuck together.

If an error occurs while testing out an address line the following error message is displayed:

```
Control Store Address line stuck at address X'xxxx' while
testing bit y
```

If an error occurs while testing out pairs of address lines the following error message is displayed:

```
Control Store address lines stuck together at address X'xxxx'
while testing bit y
```

Subphase 3 tests out the 4096 locations of the Writeable Control Store. The pattern it writes to the location is based upon the location number.

If the 80 bit pattern sent is different then the 80 bit pattern received the following error message is displayed:

```
Error occurred at the following WCS location X'xxxx' Bits in
error are y
```

4.1.3.5 Phase 5 - 29117 Internal Registers

Phase 5 tests the data and addressing path to all of the Internal Registers for the 29117 Microprocessor. Phase 5 also tests the tests the bits of the 32 (16-bit) Internal Registers, 8 bit Status Register, and 16 bit Accumulator of the 29117 chip. This test single steps two instructions to write to the register by executing two instructions to write to the Interrupt Register. The first instruction sources the test pattern from the immediate field to the D-Bus. The next instruction Latches the D-Bus into the 29117 ALU register file. The read back path requires the instructions to be loaded into the writable control store before single stepping. The first instruction tells the ALU to put the contents of

the specified register onto the YBUS output. The data is put on the immediate field read back via the execution register. This has 3 subphases.

Subphase 1 tests the data lines to the 29117 register by writing the diagnostic pattern set to register 0:

If an error occurs while writing a pattern to and reading a pattern from register 0 the following error is printed:

```
Datalines error while testing 29117 register  R'x'.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'
```

Subphase 2 tests part of the instruction to the 29117 Register. This is done by writing to each register its number and then reading back the register's value.

If the pattern written to the first 32 internal register's is different, then the pattern received from these register's the following error is printed out:

```
Address lines Error on 29117 register  Rx.
Expected: X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

If the pattern written to the accumulator is different, then the pattern received from the accumulator the following error is printed out:

```
Data mismatch while testing bits in accumulator.
 Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

If the pattern read back from the Status register is not the same as the pattern written to the status register (except for the zero bit being clear) the following error is printed out:

```
Data mismatch while testing bits in status register.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 3 tests the bits of the 32 internal registers, the accumulator, and the status register by writing the following patterns to each of these registers:

```
X'5555', X'FFFF', X'AAAA', X'0000',
```

If the pattern written to one of the internal registers is different then the pattern received from this register then one of the following error messages is printed out:

```
Data mismatch while testing bits in 29117 register.
 Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.

Data mismatch while testing bits in accumulator.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.

Data mismatch while testing bits in status register.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

4.1.3.6 Phase 6 - Interrupt Generation Test

The RBE can interrupt the JCP on LEVEL 3.

This phase tests all the interrupt vectors reserved for the RBE cards. Interrupts are used extensively by diagnostics that run microcode. The interrupt is used as a handshake from the bit slice machine to the JCP to say that the code has finished executing. Interrupts are also used in runtime to support Picking.

An interrupt is generated at each supported vector. Interrupts should occur at addresses 310, 314, 318 and at 31C. One of the following errors may be reported in this phase.

```
Interrupt occurred, but at the wrong vector.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.

TimeOut: Interrupt Did NOT occur.
```

If the TimeOut error is reported 4 times it is a good indication that the interrupts are not being generated at all. A spurious interrupt when Acknowledge is not jumpered indicates that the interrupts are jumpered on the right pins. For example a LEVEL 4 interrupt is generated instead of a LEVEL 3.

Note: Interrupts must be working before trying to run any microcoded diagnostics. Otherwise the following error may occur every time microcode is executed.

```
HA processor timed out -- PC = ( current program counter ).
```

4.1.3.7 Phase 7 29110 Microsequencer and Condition Code Multiplexer

The 29110 Microsequencer provides program flow control for the PS 390. To test the sequencer sequencer commands are loaded into the Execution Register and executed in

single step mode. The Control Store is then read back to determine if the sequencer's operation resulted in the correct program counter value. The following Sequencer Instructions are Tested:

- JMPI jump immediate
- CONT continue
- JMPV jump vector data
- CJI.cc
- LDCTI
- RICT
- JSRI.Z
- RFCT

Phase 7 tests the 29110 microsequencer. The 29110 has a program counter (PC) which is 12 bits wide. This phase has 7 subphases.

Subphase 1 tests to see if we can load the PC with an address, and then increments and reads back the PC.

If an error occurs the following message is printed out:

```
Unable to Read (set, or increment) the 2910 program counter
```

Subphase 2 tests to see if we can increment the PC from x'FFF' to x'000' and then tests to see if we can set each bit of the 12 bit PC.

If the PC did not increment the following error message is printed out:

```
Microcode program counter not incrementing.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

If the PC incremented but not to x'000' the following error message is printed out. This could indicate a problem with the CONT instruction.

```
Microcode program counter cannot hold a zero.
Expected:  X'0' Received X'yyyy' Bits in Error: X'zzzz'.
```

When testing each bit of the PC the PC is loaded with a new value and incremented. If the returned value of the PC is not incremented the following message is printed out. This could indicate a problem with the CONT instruction or a address lines error.

```
Microcode program counter not incrementing.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

When testing each bit of the PC, if the returned PC value is an unexpected value the following message is displayed:

```
Microcode program counter bit test error.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 3 tests out the continue instruction. This is done by single stepping a continue instruction. If the value to the PC is not properly incremented by the continue instruction the following error message is displayed.

```
Unable to execute the Continue Instruction
```

Subphase 4 tests out the Jump on Vector Ram address instruction. This is done by single stepping a Jump on Vector Ram address instruction. If the new PC value is incorrect the following error message is displayed:

```
Unable to execute the Jump to VEC. RAM address instruction.
```

Note: Currently there is a known bug with this test. This error will be reported but should be ignored until further notice.

Subphase 5 – tests the conditional jump instructions. The first 4 tests are done by single step execution and the remaining 6 instructions are executed in microcode. These instruction tests will now be described in their proper order.

The following condition codes are tested:

- True
- False
- Zero, NZero
- Negative
- Carry
- Overflow
- CT
- FR Fifo stack bus control ready
- NPPL attention bit

The FSBC (Fifo Stack Bus Controller) ready bit (*INFSBCREADY) is tested for a true condition. This means the diagnostic tests for Bit = 1 or Bit = 0. This condition should exist after resetting the PS 390. The test is done by resetting the PS 390 and then attempting to jump on the FSBC being true. If the improper condition exists and the jump is not taken the following error message is displayed:

```
Unable to branch properly on the FIFO STACK BUS CONTROLLER
condition.
```

Errors reported in RBED1A the Endpoint Graphics Pipeline diagnostic may be related to this condition code.

The Pixel Processor Loader is tested for a false condition. This condition should exist after resetting the PS 390. The test is done by resetting the PS 390 and then attempting to jump on the PPLATTN bit being false. If the improper condition exists and the jump is not taken the following error message is displayed:

```
Unable to branch properly on the NOT PIXEL PROCESSOR condition.
```

The PS 390 is reset by toggling bit 14 in the Maintenance register. This test sets up a zero condition and then attempts to jump on this condition. If the jump is not taken the following error message is displayed:

```
Unable to branch properly on the Z ( ZERO ) condition.
```

This test sets up a not zero condition and then attempts to jump on this condition. If the jump is not taken the following error message is displayed:

```
Unable to branch properly on the NOT Z (ZERO) condition.
```

This test checks out to see if the 29117 can properly increment one of its internal registers. The reason this is done is because the remaining 5 tests will be incrementing this internal register as part of their tests. If this test fails the following message is displayed:

```
29117 ALU is not functioning properly.
```

This test sets up a negative condition by loading a 29117 register with x'FFFF' and then latching in the negative condition. A jump is then attempted off of this condition. If the jump fails the following error message is displayed:

```
Unable to branch properly on the N ( NEGATIVE ) condition.
```

This test sets up a carry condition by loading a 29117 register with x'FFFF' and then incrementing the register. A jump is then attempted off of this condition. If the jump fails the following error message is displayed:

```
Unable to branch properly on the C (CARRY) condition.
```

This test sets up an overflow condition by loading a 29117 register with x'7FFF' and then incrementing the register. A jump is then attempted off of this condition. If the jump fails the following error message is displayed:

```
Unable to branch properly on the O (OVERFLOW) condition.
```

This test sets up a CT true condition by loading a 29117 register with x'FFFF', incrementing the register, and then doing an ALU test on the carry condition. A jump is then attempted off of the CT true condition. If the jump fails the following error message is displayed:

```
Unable to branch properly on the CT condition.
```

This test sets up a NOT CT condition by loading a 29117 register with x'00FF', incrementing the register, and then doing an ALU test on the carry condition. A jump is then attempted off of the NOT CT condition. If the jump fails the following error message is displayed:

```
Unable to branch properly on the NOT CT condition.
```

Subphase 6 tests the 29110 internal register. This is done in microcode. The maximum value the register can hold is loaded and then it is decremented until it reaches a zero value. If an error occurs the following message is displayed:

```
Internal Register is not working properly.
```

Subphase 7 tests the 29110's stack. This is done in microcode. This test relies on a working internal register. An address is pushed onto the stack and then popped off with a jump to subroutine instruction. If an error occurs the following message is displayed:

```
Internal Stack Register is not working properly.
```

4.1.3.8 Phase 8 -29117 Microprocessor Instruction Confidence

Not Yet Implemented.

Errors in the several other microcoded phases may indicate a malfunctioning 29117 ALU processor.

The 29117 microprocessor is the 16-bit ALU for the PS 390. Testing of the ALU includes performing all arithmetic and logical functions, and checking the results.

4.1.3.9 Phase 9 - Scratch RAM

The Scratch RAM for the raster back end is 16-bits by 2048 words. Access to the Scratch RAM is via the Bitslice processor. Reading the scratch RAM requires the use of the ACC register in the 29117 processor. Data is written to the ACC via the same path described in the Arithmetic Logic Unit Register test. It is then put on the Y-BUS and placed in the YTEMP register of the Y to D bus transceiver. The next instruction places address of the scratch Ram on the immediate field and the data is written to that location over the D bus.

This test addresses from 0 to 2047, reading, writing, and checking test patterns at each location.

Subphase 1 checks all of the Scratch Ram.

If the 16 bit pattern sent is different then the 16 bit pattern received the following error message is displayed.

```
Datalines error writing to Scratch Ram bits.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 2 tests out the 11 address lines to the Scratch Ram. First each address line is tested to see if is stuck high or low, then each possible pair of address lines are tested to see if they are stuck together.

Address line errors are reported in one or both of the following ways.

```
Scratch Ram Address line stuck at address X'xxxx' while testing
bit y.
```

```
Scratch Ram address lines stuck together at address X'xxxx'
while testing bit y
```

Subphase 3 tests out the all 2048 locations of the Scratch Ram. This is a microcode subphase. The first test write location to location. Errors in this subphase result in the following error message.

```
Error occurred at the following Scratch location X'xxxx' Bits in
error are y
```

A timeout error may occur if the interrupts are not generated properly.

Subphase 4 is a random number test on scratch Ram. Random numbers are written to all the locations at full speed using microcode. The data is read back and compared for errors. Errors in this subphase result in the following error message.

```
Random pattern error found in Scratch Memory at ( SCRAM address )
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
```

4.1.3.10 Phase 10 - Vector Ram

Access to the Vector RAM is via the Bitslice processor. A data and address lines test is run on the RAM.

Phase 7 tests the Vector Ram. The Vector Ram has 7 pages, each page has 256 words. Each word is 12 bits wide. This has 4 subphases.

Subphase 1 tests out the data lines to the Vector Ram.

If the 12 bit pattern sent is different then the 12 bit pattern received the following error message is displayed.

```
Datalines error writing to Vector Ram bits.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 2 tests out the 8 bits of address lines.

If an error occurs while testing out an address line the following error message is displayed:

```
Vector Ram Address line stuck at address X'xxxx' while testing
bit y
```

If an error occurs while testing out pairs of address lines the following error message is displayed:

```
Vector Ram address lines stuck together at address X'xxxx'
while testing bit y
```

Subphase 3 tests out the 3 bits of page lines.

If an error occurs while testing out an address line the following error message is displayed:

```
Vector Ram Page line stuck at page address X'xxxx' while
testing bit y
```

If an error occurs while testing out pairs of address lines the following error message is displayed:

```
Vector Ram Page address lines stuck together at address X'xxxx'
while testing bit y
```

Subphase 4 tests out the 7 pages of 256 words of the Vector Ram. The pattern it writes to the location is based upon the location number.

If the 12 bit pattern sent is different then the 12 bit pattern received the following error message is displayed.

```
Error occurred at the following SCR location X'xxxx' Bits in error
are y
```

4.1.3.11 Phase 11 - Function Lookup Table

The EPROM Lookup table for the Raster Backend portion of the PS 390 is 16-bits by 65536 (or 64K) words. Access to the Lookup table is via the Bitslice processor. This test addresses from 0 to 65536, reading the EPROM at each location. A check sum is performed and compared to the last location in the EPROM where a checksum is burned into the prom. If the two checksums do not agree the following error message is reported.

```
Checksum error reading Function Look Up Table Prom.
Computed checksum: xxxx Checksum in FLUT: yyyy.
```

When the computed checksum is correct the proper value.

4.1.3.12 Phase 12 - AMD 29517A Multiplier

Not Yet Implemented.

This is a 16 by 16 bit parallel multiplier. Access to the multiplier is via the Bitslice Processor. This test multiplies different test patterns together and verifies the results.

4.1.3.13 Phase 13 -Common Bus Direct Memory Access (DMA)

A DMA from the RBE bit slice requires data to be loaded into a 16 bit common bus data register, and 22 bit address to be loaded into two address registers. The DMA hardware is tested in three subphases. The first two are single stepped and the third is a full speed microcoded phase.

Subphase 1 tests out the data register.

A 16 bit set of diagnostic patterns are written to a location in mass memory. The microcode program waits for the *cbbusy signal to go low after requesting a common bus access. If the microcode does not receive the *CBBUSY low signal, the following errors are reported.

```
HA processor timed out -- PC = X'yyy'.
Timeout occurred waiting for cbbusy signal.
```

The JCP then reads location X'200000' to verify that the data was written correctly. DMA write errors are reported as follows.

```
   Datalines Error writing data from RBE to mass memory.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
```

Then the RBE Bitslice reads the pattern back via the DMA hardware and reports any read back errors as follows.

```
Datalines Error reading data from mass memory to RBE.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
NOTE: Assumes data was correctly written to mass memory.
```

Subphase 2 tests out the addressing path to the Common Bus. Test data is written via the RBE DMA hardware to each location, setting one address bit at a time. The JCP then reads the location to verify that the pattern arrived at the proper location. The

LSB address lines are all tested. However, to test all of the upper address lines mass memory cards from 200000 to 600000 must be in the system.

Address line errors may be reported in the following way.

```
RBE to common bus addressing error testing address X'xxxxxx'.
```

Subphase 3 is a full speed read/write test of the DMA hardware using random number test patterns. A packet of 8 Random numbers are written to Mass memory and read back and compared. Then another packet of 8 patterns are written and so on until 2000 packets of 8 are transfered. When an error is detected during the test, the test is interrupted to report that error. The mass memory location is examined to see if the data was written properly. If the expected pattern does not appear in mass memory the following error is reported.

```
Data mismatch at location X'yyyyyyy' after full speed common bus
write operation.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
```

If the data in Mass memory is correct then it is assumed the error occurred on the readback and the following error is reported.

```
RBE read back data mismatch at location X'200000' testing full speed
random number DMA transfer.
Expected: X'xxxx'  Received X'yyyy' Bits in Error: X'zzzz'.
 NOTE: Mass Memory contains correct data.
```

Parity errors that occur in mass memory during this full speed transfer are counted and any parity errors are reported as follows.

```
Parity interrupt during common bus full speed read/write test.
```

Note: The modify command will allow you to select a different Megabyte of mass memory to perform these tests on. This phase accesses 2 separate memory cards (or banks) by alternately writing a bit pattern to each megabyte. In order for this phase of the diagnostic to work properly, the system must have at least 2 Megabytes of memory on the Mass Memory Board.

4.2 Endpoint Pipeline Diagnostic - RBED1A

The Endpoint Pipeline is comprised of custom VLSI chips, that are also used on the Shadowfax project. The pipe includes the Input FIFO Stack Bus Controller, the Delta/Depth Cue Calculator, the EndPoint FSBC and the Color FSBC. In addition, the RBE contains an interface from the pipeline to the Pixel Processor Data Bus which allows the data to be sent to the pixel processor. A diagnostic path allows the data to be read back to the Master Bitslice for verification. Otherwise the Pixel Processor Data path leads to the VLSI Pixel Processor array. The Endpoint Pipeline has the following components:

(a) Input FSBC controller
(b) Input FSBC Data Register
(c) Input FSBC
(d) Delta / Depth Cue Calculator
(e) Depth Cue RGB
(f) Output FSBC Color RGB
(g) Output FSBC Endpoint
(h) FSBC to Pixel Processor Data Bus
(i) Pixel Processor Data Bus readback path via D-Bus

4.2.1 Testing Strategy

The strategy for testing the Pipeline is to first test the Input FSBC and FSBC data registers by sending a set of patterns through the pipe in the transparent mode. This tests the Endpoint FSBC but not the Color FSBC. The Pipeline control signals can be verified at this point. Followed by testing for the Delta/Depth Cue Calculator and Output FSBC by sending a set of test vectors through the pipe.

Phase 1 - RBE Common Bus Maintenance register test.
Phase 2 - Transparent mode test sending zeros through the pipe.
Phase 3 - Full pattern test in transparent mode to test readback paths.
Phase 4 - Functional test on the Delta/Depth Cue Calculator.

4.2.2 Description of Tests

4.2.2.1 Phase 1 - Common Bus Maintenance Register

Phase 1 tests out the Maintenance Register located at X'FFF030'. Phase 1 has 2 subphases.

This phase tries to read and write to the Maintenance Register. If a bus error occurs, a diagnostic interrupt handling routine detects it and an error message is reported.

Subphase 1 of Phase 1 attempts to read the maintenance register. If a bus error happens during the reading of the maintenance register the following error occurs.

```
Common bus read error for Raster Back End MR:
        (Maintenance register's contents).
```

Subphase 2 of Phase 1 attempts to write to the maintenance register and to set and reset various bits of the maintenance register.

If a bus error occurs during the writing of the maintenance register the following error occurs.

```
Common bus write error for Raster Back End MR: ( PS 390
        Maintenance register's contents ).
```

4.2.2.2 Phase 2 - Transparent mode test sending zeros through the pipe

Phase 2 of RBED1A attempts to send any data down the graphics pipeline. Subphase 1 of Phase 2 sends the transparent command to the pipe and writes zeros in the x,y,z and w. The data is read back and stored in scratch memory.

There are several handshaking signals that the microcode expects to see while writing to the pipe. If the microcode times out while waiting for one of these signals one of the following errors will be reported after the timeout error is reported:

```
Microcode timed out waiting for FSBC ready bit.
```

The FIFO Stack Bus Controller Ready bit did not go low after card reset. This bit should go low before the microcode will write any data to the pipe.

```
Microcode timed out waiting for PPL attention bit.
In.PACK not acknowledged by *PPLATTN.
```

After data is written to the FSBC intermediate registers an IN.PACK signal is sent. The *PPLATTN bit should acknowledge this signal.

```
FSBC busy after PPL.PACK signal sent.
```

The FIFO Stack Bus Controller Ready bit did not go low after data was sent. This might indicate that the PPL.PACK signal may not have been recognized.

The data stored in scratch memory is checked for errors. First the command word is check to verify that it came through the pipe intact. Errors in the readback of the command word are reported as follows:

```
Unable to write Transparent Command Word to FSBC.
Expected: X'8060C00'  Received X'yyyyyyyy'.
```

This is considered a Fatal error and will prevent any other tests from executing.

Non zero values read back for the X,Y and Z locations are reported.

```
Unable to write a Zero through the Graphics Pipe.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 2 of Phase 2 sends all the possible command words through the pipe. Command words are sent through all the chips in transparent and should retain the original value when reading values at the bottom of the pipe. The following command words are tested.

- X'80400006 enable delta calculator
- X'80400004 disable delta calculator
- X'80400018 enable dot mode
- X'80400010 disable dot mode
- X'80400060 enable depth cueing
- X'80400040 disable depth cueing

Errors are reported as follows:

```
Error sending Command words through graphics pipe.
Expected: X'8040xxxx'  Received X'yyyyyyyy'.
```

4.2.2.3 Phase 3 - Full pattern test in transparent mode

Phase 3 of RBED1A runs a full diagnostic pattern test throught the pipeline in transparent mode. This may result in some occasional signal errors detailed in subphase 2. These should not be ignored. Check the timing pals for these signals. Data errors may also be reported. Occasional data that is incorrect should not be ignored. This test is a critical timing test. Several repetitions of this phase is suggested.

```
Error sending patterns through the Graphics Pipe.
Expected X: X'xxxxxxx'  Received X: X'yyyyyyyy'.
Expected Y: X'xxxxxxx'  Received Y: X'yyyyyyyy'.
Expected Z: X'xxxxxxx'  Received Z: X'yyyyyyyy'.
```

Note: The bits in error reported are not correct.

Note: There is no path to read the W component of the vector.

4.2.2.4 Phase 4 - Functional test on the Delta/Depth Cue Calculator.

Not yet implemented. A full functional test of the DDCC will take some time to develop.

4.3 Frame Buffer Diagnostic - RBED2A

The Raster Backend Frame buffer is a modified version of the Shadowfax frame buffer. The Frame buffer components include:

(a) 1024 × 1024 × 48 Image bit planes
(b) 1024 × 1024 × 8 Window/Valid planes
(c) Frame Buffer Memory Controller

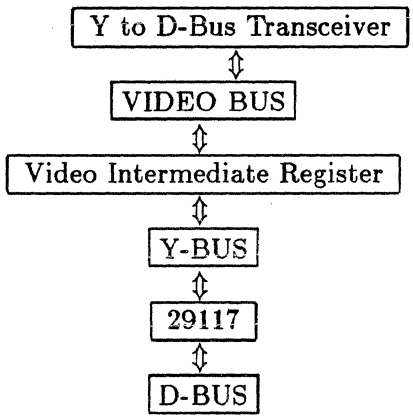
4.3.1 Testing Strategy

The strategy for testing the Frame Buffer is to first test the video control register, the color look up table, the Pixel processor data register and then the pixel processors, the Scan line buffer and then the DACs. The pixel processor test relies on visual feedback. The scan line buffer controller and interface are then tested by writing values to the Frame Buffer and reading them back. Next the frame buffer memory is tested and finally the DACs.

Phase 1 - Video Control Register Test
Phase 2 - Color Look up table Test
Phase 3 - Pixel Processors (16)
Phase 4 - Pixel Processor ONCOUNT, TOTCOUNT and INTENSITY registers
Phase 5 - Frame Buffer and Scanline Buffer Memory
Phase 6 - DAC test.
Phase 7 - Visual Debugger. You can draw boxes.
Phase 8 - YASD Debugger.

4.3.1.1 Phase 1 - Video Control Register Test

Phase 1 tests out the PS 390's Video Control Register by doing a write and read of the register. Each bit is toggled. Flashing of the display is expected in this test. The contents of the video control register are read back into scratch ram location 1 over the Y-Bus through the Y to D transceiver. Therefore this path must be verified before testing the video card.



Subphase 1 of Phase 1 verifies the contents of the video control register after it has been written. Bit 11 of the register cannot be read and will always be high. Other than this bit, if a bit appears to be stuck the following message is reported.

```
Read Write Error testing the video control register.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.
```

4.3.1.2 Phase 2 - Color Look Up Table

For each R, G, and B value there is a Color Look-up Table. The Color Look-up tables are located in the DACs. There is a read/write path via the video intermediate register for the data. The color look up table address register can be read back via the signature readback path. The addressing and data paths for Red, Green and Blue Look Up tables are verified and then the CLUT is filled.

Subphase 1 of Phase 2 checks the addressing path to the Color look up tables. An address is written to the Video Intermediate Register and the Color Look Up Table Address register is enabled. This puts the address on the %VLABUSD inputs to the DACS. The %VLABUSD bus is read back via the signature readback path into scratch

ram location 1. Bit 0-7 represent the address sent to the RED DAC and bits 8-15 represent the address to the GREEN DAC. If an error occurs the following message is reported.

```
Read Write Error testing the Color LUT Address register.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 2 of Phase 2 verifies the 8 bit data path to the Red, Green and Blue look up tables. Data is written to the the Video Intermediate register and the CLUT_Red, CLUT_Green, and CLUT_Blue lines are enabled. The data is read back for verification. If an error is detected one of the following messages will be reported.

```
Read Write Error testing the Color LUT Red register.
Expected: X'00'  Received X'yy' Bits in Error: X'zz'.

Read Write Error testing the Color LUT Green register.
Expected: X'00'  Received X'yy' Bits in Error: X'zz'.

Read Write Error testing the Color LUT Blue register.
Expected: X'00'  Received X'yy' Bits in Error: X'zz'.
```

Subphase 3 of Phase 2 is a microcode test that fills the color look up table with an incrementing pattern and reads it back full speed to verify the contents. The path that is used is the same path described in subphase 1 and 2 with the exception that scratch ram is not used, the data is latched into the ALU for comparison. Errors in this subphase are reported as follows:

```
Error in the Color Look Up Table from the Red DAC.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.

Error in the Color Look Up Table from the Green DAC.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.

Error in the Color Look Up Table from the Blue DAC.
Expected: X'0000'  Received X'yyyy' Bits in Error: X'zzzz'.
```

4.3.1.3 Phase 3 - Pixel Processors (16)

The Pixel Processor Array consists of 16 identical processors with eight processors located on each of the two RBE cards. The Pixel Processors update Frame Buffer memory. The

primary purpose of the PPA is to draw anti-aliased lines, implement polygon fill, and block transfers. The PPA also provides read and write access to the Frame Buffer from the RBE Bit Slice Processor. Testing is done by displaying visual patterns which verify the operation of the Pixel Processors and their packet registers.

Initializing the pixel processor array takes several steps. Each step is recorded in scratch memory location 50 as it completes. If the microcode times out during initialization of the PPA the following error message is reported along with the last step completed.

```
Unable to initialize Pixel Processor Array. Last step = <0-9>.
```

Subphase 1 of Phase 3 initializes the color look up table and pixel processor array.

The microcode depends on several handshake signals to initialize and write to the pixel processor registers. If the microcode is stuck in a loop waiting for the appropriate signal response a time out error will occur and one of the signal error messages may be reported.

```
HA processor timed out -- PC = ( current program counter ).
```

After a PP.PACK signal is sent to write to the pixel processors, the μcode waits for the PPREADY (bit 12 in STATUS reg) to go low. PPACTIVE signal will then go high and the flush routine waits for PPACTIVE (bit 11 in STATUS reg) to go low before attempting to write to another pixel processor.

```
Timeout waiting for NPPREADY signal to go low in PP flush routine.
```

```
Timeout waiting for PPACTIVE signal to go low PP flush routine.
```

The following error is very similar in nature but it occurs while writing to individual pixels on the screen to complete a scan line. This is a faster routine and does not wait for the PPREADY signal to go low.

```
Waiting for PPACTIVE signal to go low in draw loop.
```

A test is performed to verify the vertical blank signals writing to the frame buffer. If the vertical refresh does not occur one of the following errors is reported.

```
waiting for NOT vertical blank in FRAMEWAIT.
```

```
Waiting for vertical blank to go high in FRAMEWAIT.
```

Subphase 2 of Phase 3 tests tests the operation of the individual pixel processors. A large 4 by 4 grid is displayed, with each element representing by position, the respective pixel in a 4 by 4 grid of pixels that each pixel processor controls. (See Figure 4.1) A pattern of red dots is expected in each element, indicating the operation of an individual pixel processor. The decoding of rows and columns into Pixel Processor number and card is shown in Table 4.1. The boxes are numbered from 0 to 15 starting at the top left hand corner, counting across and down. The box number corresponds to the pixel processor number.

PP0	PP1	PP2	PP3
PP4	PP5	PP6	PP7
PP8	PP9	PP10	PP11
PP12	PP13	PP14	PP15

Figure 4.1: Pixel Processor Assignment

4.3.1.4 Phase 4 - Pixel Processor Register Test

Phase 4 of RBED2A tests out a few of the Pixel Processor registers. There are 64 internal registers in each Pixel Processor. This phase performs a visual test for a few of these registers. There are three subphases in phase four. The operator must hit a key on the debug terminal to proceed from one subphase to the next.

Use the MODIFY command to change the PPMASK. PPMASK is a register that will turn each Pixel Processor on or off depending on whether the corresponding bit is set in PPMASK. For example, a PPMASK of X'FFFF' turns all the Pixel Processors on. A value of X'0080' turns on only Pixel Processor number 7. A modified PPMASK will affect all three subphases of Phase 4.

Subphase 1 of Phase 4 tests the TOTCOUNT register. The value in TOTCOUNT

Table 4.1: Pixel Processor Row and Column Decoding

Pixel Processor Row and Column Decoding			
Row	*Column*	*Number*	*Card*
0	0	0	204485-600
0	2	1	204486-600
1	0	2	204485-600
1	2	3	204486-600
2	0	4	204485-600
2	2	5	204486-600
3	0	6	204485-600
3	2	7	204486-600
0	1	8	204485-600
0	3	9	204486-600
1	1	10	204485-600
1	3	11	204486-600
2	1	12	204485-600
2	3	13	204486-600
3	1	14	204485-600
3	3	15	204486-600

controls the length of the scan line. TOTCOUNT is incremented by one each Scanline creating a pattern increasing from top left to bottom right should be displayed. Any lines not following this pattern would be an indication that the TOTALCOUNT register is not functioning correctly.

Subtest 2 of Phase 4 tests the operation of the ONCOUNT register. The test is performed by setting TOTALCOUNT to the maximum value, then drawing a series of horizontal lines while varying the value of ONCOUNT from minimum to maximum. Set up in this way the value in ONCOUNT controls the length of the Scanline. A pattern increasing from top left to bottom right should be displayed. Any lines not following this pattern would be an indication that the ONCOUNT register is not functioning correctly. Spurious blue dots or lines against the white background can be eliminated by changing PPMASK.

Subtest 3 of Phase 4 tests the operation of the INTENSITY registers. The test is performed by drawing blue, green, and red blocks while varying the INTENSITY from minimum to maximum intensity. The color intensity should increase from left to right. Failure to produce a smooth and consistently increasing pattern would be an indication that the INTENSITY registers are not functioning correctly.

4.3.1.5 Phase 5 - Frame Buffer and Scanline Buffer Memory

The Frame Buffer memory is organized as a 1024 x 1024 x 52-bit memory that drives the raster display. The memories are 256K video RAMs that are dual ported. The random port is controlled by the Pixel Processor Array to allow reading and writing the Frame Buffer. The serial port allows data to be scanned and sent to the Video Controller which then drives the raster display. There are two banks in the Frame Buffer. they are called Bank A and Bank B. This phase tests both banks by default. The MODIFY command will allow the operator to disable one or both of the banks.

The data path to the Frame Buffer is tested by writing a 16 bit test pattern to one scan line in the Frame Buffer and then reading it back.

The entire Frame Buffer memory is tested using complementing eight bit patterns, written to the blue, green, red, and window locations for each pixel. The data is complimented every 4th pixel so that vertical stripes should appear on the screen. After completely filling memory, data is read via the Scanline buffer and checked for errors. If any frame buffer memory errors are found, the following message is displayed:

Note: The vertical stripe should appear in black and white or as a shade of gray. If the color is wrong (e.g. blue and white stripes) then this indicates a problem with the background color registers.

```
Frame Buffer memory error.
Expected Green/Blue : X'ggbb'  Received Green/Blue : X'ggbb'
Expected Window/Red : X'wwrr'  Received Window/Red  : X'wwrr'
Xaddress : X'xxxx'     Yaddress : X'yyyy'
```

Where the data is received in two packets Green/Blue and Window/Red. If there appears to be a consistent bit stuck in one of these packets the problem is most likely in the read back path. Occasionally on power up one may see a single error reported in this phase, repeat the phase and it will go away.

Handshaking signal errors may also be reported as documented in phase 3 of this diagnostic.

4.3.1.6 Phase 6 - DAC Test

This phase is not yet implemented. The plan, however, is to load the CLUT with all white except for the test bit, which will be black. The frame buffer is then filled with the test pattern which results in a black screen. In the case of an error there will appear white dots indicating that one of the outputs of the DAC is stuck.

4.3.1.7 Phase 8 - Visual Debugger (optional)

Phase 5 Allows you to draw little boxes all over the screen. Command options are as follows:

(M) Modify Box Parameters.
(R) Reset HA board.
(I) Initialize Pixel Processors.
(C) Load Color Lookup Table.
(D) Draw a Box.
(Q) QUIT.

To use this debugger you must first (I) Initialize the Pixel Processors. Then (C) Load the Color Lookup Table and (D) Draw a box. This will fill the screen with white. Signal errors as described in Phase 2 can be reported for each one of these steps. A (R) reset of the HA board will require re-initialization of these first three steps.

Use the (M) command to modify the box parameters. A number must be entered for each prompt or a zero will default. The values you can specify are as follows:

```
X address < 400 - 0 > specifies the top left corner of the box.
Y address < 400 - 0 > specifies the top left corner of the box.
X size    < 400 - 0 > specifies the length of the box
Y size    < 400 - 0 > specifies the width of the box
PPMASK    < FFFF - 0> specifies the number of Pixel Processors involved
WINDOW    < FF - 0 >  08 is the preferred value.
RED       < FF - 0 >  Red intensity
GREEN     < FF - 0 >  Green intensity
BLUE      < FF - 0 >  Blue intensity
```

4.4 MPLSD0B: PLS Analytic Diagnostic

MPLSD0B uses data analysis and signature analysis to test the circuitry on the PS 350 Pipeline Subsystem (PLS) card (E&S #204143-100) and the interface to the Refresh Buffer (RFB) card on the PS 350 or the Raster Backend Card Set on the PS 390. Data analysis uses the results of calculations from each section of the card and compares these results with expected results to determine if the circuitry is functioning properly. Signature analysis is used in areas where data paths are not provided.

MPLSD0B performs signature analysis on five PLS card nodes. These nodes are organized so that they correspond with subsections of the PLS card. Phases 1 and 9 use different types of signature stimulus to test the same circuitry. Phase 1 breaks all feedback paths to ACP and clock control and sends a simple vector list down the pipe to test handshaking between the ACP, PLS, and shadow pipeline. Phase 9 sends a complex vector list containing multiple viewport changes down the pipe.

The remaining phases of the diagnostic perform data analysis on the calculation components of the card.

Loop on error is only used in the pipeline configuration register test in Phase 1. The other phases are unable to loop on error because they send vectors down the pipeline and it is undesirable to loop on the entire vector list. The optional Phase 11, the PLS debugger, provides a better way to loop on error. It allows the user to set up the pipeline and specify and probe the vector or vectors that are continuously sent down the pipe when the vector loop command is implemented. The pipeline can be set up to simulate the phase that fails. Set-up information is available in the phase descriptions.

Note that each diagnostic phase gives meaningful results only if all previous phases run successfully without errors. Each phase depends on the results of the previous phase.

4.4.1 Functional Description

MPLSD0B consists of eleven phases:

Phase 1 Tests PLS status, the pipeline configuration register, ACP/PLS handshaking, and the shadow pipeline signatures.
Phase 2 Tests MULBUS paths to the PLS and FIFO.
Phase 3 Tests the Block Normalizer.
Phase 4 Tests the perspective divider circuit.
Phase 5 Tests the IYX data path.
Phase 6 Tests the viewport and intensity registers.
Phase 7 Tests viewport and intensity multipliers.
Phase 8 Tests the PLS clipping circuit.
Phase 9 Performs PLS signature tests.
Phase 10 Tests the PLS/Refresh Buffer interface.
Phase 11 Is an optional phase that implements the PLS debugger.

A functioning 4K ACP card (E&S #204133-100) is required for MPLSD0B to execute properly.

The following default values are used when the Diagnostic Operating System loads this diagnostic.

- The default file name that contains the known signatures for the diagnostic MPLSD0B is PLSSIGA.TXT.
- Ten signatures are gathered on each node before determining if a signature is stable.

The PLS Diagnostic routines for the PS 390 are the same as the diagnostics for the PS 350 with the exception that Phases 1, 8, 9 and 10, which have been modified for the PS 390.

Phase 1 determines whether or not the system has a refresh buffer or a Raster Backend. If neither one is present the following error message is displayed:

```
Unable to read a 350 or 390 Maintenance Reg.
```

In Phases 8 and 9, microcode is loaded into the PS 390 to consume bytes coming down the pipeline from these tests. The microcode waits for the FIFOREADY bit to be set before reading in a byte of data.

In Phase 10, a full set of test patterns are transferred from the PLS card to the Raster Backend. Subphases 1 and 3 send data to the PS 390. Subphase 1 loads bytes directly

through the IYX data register to the PS 390. Subphase 3 sends IYX vectors down the pipeline to the PS 390. Both subphases have microcode running in the PS 390 that reads data from the PS 390's input Fifo and writes it into the scratch ram. The microcode waits for the FIFOREADY bit to be set before reading in a byte of data. Both subphase's microcode will try to read in a specific amount of bytes (54 for subphase 1, 30 for subphase 3) before terminating. If a sufficient number of bytes are not received from the PLS by the PS 390 the JCP will timeout with the following error message:

```
HA processor timed out -- PC = ( current program counter ).
```

In both subphases the patterns are then read out of scratch ram by the JCP and are compared with the expected data.

Subphase 1 of Phase 10 performs an IYX data test on the interface to the PS 390.

If a data mismatch between the expected data and the received data form the PS 390, the following error message is displayed:

```
Data error in PLS/PS 390 Interface.
Expected:  X'xxxx' Received X'yyyy' Bits in Error: X'zzzz'.
```

Subphase 3 of Phase 10 performs a vector data test on the interface to the PS 390. IYX data is sent down the pipeline to the PS 390:

If there is a data mismatch between the expected data and the received data from the PS 390, the following error message is displayed.

```
Vector data error in PLS/PS 390 Interface.
Vector number: xxx Section: (I or Y or X)
Expected: zzz, Received: rrrr.
```

MPLSD0B signatures are in standard hexadecimal format. Hewlett-Packard signature format is not used in this diagnostic.

4.4.2 Initialization

When the Diagnostic Operating System loads MPLSD0B, this phase executes automatically and the following message is displayed:

```
PS\350 PLS Signature Verification Diagnostic.
```

Then the diagnostic reads the root node signature file "PLSSIGA.TXT". If an error occurs the following message is displayed:

```
Error ee occurred in reading signature table file: PLSSIGA.
```

where ee is one of the possible error messages of the Diagnostic Operating System. If the file is read successfully, the diagnostic checks that stimulus program names in the file all have the name PLSSTM0A. If all of them are not named PLSSTM0A, the following message is displayed:

Invalid stimulus program names stored in file. Do not proceed with this diagnostic.

Do not execute the diagnostic any further (unless you have an alternate file) because this version of the diagnostic expects all stimulus programs to be PLSSTM0A.

4.4.3 Parameter Modifications

Five of the signature verification parameters may be changed using the "M"odify command. After the cursor ">" appears, a menu is displayed with the following options:

```
Available Options
<CR>  =  Exit modify phase.
  0   =  Display this menu.
  1   =  Modify frame count.
  2   =  Create a new signature table.
  3   =  Display the current signature table.
  4   =  Specify an alternate file name.
  5   =  Enable optional phases.
Enter modify option::
```

Entering a RETURN (<CR>) causes the diagnostic to exit Modify. Enter a 0 to redisplay the menu of available options. Entering any number other than 0 → 5 causes the message:

```
Invalid Option
```

and the system prompts for another Modify option. A number from 0 to 5 causes the associated option, detailed below, to be selected.

4.4.3.1 Option 0

Option 0 displays the options menu.

4.4.3.2 Option 1

Option 1 sets the signature frame count. It allows the user to change the number of frames performed before determining if a signature is stable. The system prompts:

```
Current number of frames performed: ffff
Enter number of frames to be performed (minimum = 3)
```

where ffff is the number of frames currently performed. To change the signature frame count, enter the number of frames to be performed (from 3 to 32000).

4.4.3.3 Option 2

Option 2 creates a file on the diskette that contains the gathered signatures.

The diagnostic prompts:

```
Do you wish to use alternate file name other than standard ?
```

If the user enters <CR> or "N," the diagnostic proceeds to Option 3 and the standard file name PLSSIGA.TXT is used. If "Y" is entered, the diagnostic prompts with:

```
File name:
```

Type the new name of the file for the gathered signature table. Unless otherwise specified, the diagnostic assumes an extension of .TXT and the highest version number.

Next, the diagnostic prompts:

```
Enter signature file information:
```

Enter up to 80 characters of configuration information. The system next prompts:

```
Enter board name (PLS, LGS, etc.) for node n:
```

where n is the number of the node.

The diagnostic then prompts for the device number to be connected to the specified node:

```
Enter device number (U125, U88, etc.) for node n.
```

where n is the number of the node.

The system then prompts:

```
Select a drive (<CR> for default)
```

Enter a number between 0 and 3 to specify a disk drive where a diskette is mounted.

If the diagnostic succeeds in writing the file, it displays the following:

```
File ffffffff created successfully.
```

where ffffffff is a file name.

Otherwise, the following message appears:

```
Error ee occurred in creating file: ffffff.
```

where ee is a number indicating one of the Diagnostic Operating System errors or one of the following:

```
ee = 47 means that bit map does not match directory entries.
ee = 48 means that the directory is full.
ee = 49 means that the new name and version already exists.
```

The diagnostic is then ready to receive a new Modify option.

4.4.3.4 Option 3

This option allows the contents of the known signature table for any phase to be displayed. If there is no signature file on the disk, the diagnostic reports:

```
No signature information has been gathered or read in.
```

The diagnostic then displays the following:

Signature file information: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhh

Board_Name	Device_Number	Known_Value	Gathered_Value
Phase number: n			
PLS	Uddd	xxxx	vvvv
PLS	Uddd	xxxx	vvvv
PLS	Uddd	xxxx	vvvv
PLS	Uddd	xxxx	vvvv
PLS	Uddd	xxxx	vvvv

where h's describe configuration information, n is the phase number, Uddd's are card and device numbers, xxxx's are the known values stored in the memory, and vvvv's are the gathered signature input values.

4.4.3.5 Option 4

Option 4 allows the user to specify an alternate file as the source of the set of known signatures. Because the diagnostic does not perform a validity check on the stimulus program names, the name is assumed to be PLSSTM0A. The system prompts:

```
Do you wish to specify an alternate file as the source
of the set of known signatures?
```

Enter a <CR> or "N" to have the diagnostic accept another Modify option. If "Y" is entered, the diagnostic prompts with the following:

```
File name:
```

Type the name of the file that contains sets of signatures to be compared with those gathered from the PLS. If the file is read successfully, the following is displayed:

```
File ffffffff read successfully.
```

where ffffffff is the name of the file read. Otherwise, the following message reads:

```
Error ee occurred in reading file: ffffffff.
```

where ee is a Diagnostic Operating System error number.

4.4.3.6 Option 5

Option 5 enables optional Phase 11, the PLS debugger.

4.4.4 Detailed Phase Description

An "(L)" to the left of an error message indicates that the error has looping capability.

4.4.4.1 Phase 1

Phase 1 reads the status of the PLS card and reports any problems, tests the pipeline configuration register, tests the signature of the handshaking lines between the ACP and the PLS cards, and tests the signature of the shadow pipeline. Phase 1 has four subphases.

Subphase 1 of Phase 1 Reads the status of the PLS card and reports any problems. This subphase also reports if the RFB card is in the system. The subphase reads status information from the upper four bits of the FIFO. Status information includes:

*MULBLO	Bit 12	If low, the Lower MULBUS cable is connected to the ACP card.
MULBHI	Bit 13	If low, the Upper MULBUS cable is connected to the ACP card.
RBEXISTS	Bit 14	If low, the Refresh Buffer card is in the system.
FIFOEMPTY	Bit 15	If low, the FIFO is empty

If the lower MULBUS is not connected correctly, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4601.1
Lower MULBUS cable not connected correctly
```

If the upper MULBUS is not connected correctly, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4602.1
Upper MULBUS cable not connected correctly
```

If the FIFOEMPTY bit is in error, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4603.1
FIFO should be empty, but it is not
```

Subphase 2 of Phase 1 Tests the pipeline configuration register. The ACP writes a full set of 36 bit patterns (see Table MPLSD0B-1) into the pipeline configuration register and reads and compares the bit patterns with what was sent.

If an error is encountered while testing the pipeline configuration register, the following message is displayed:

```
(L)  ***** MPLSDOB  ;1 - Phase 1  Error number 4604.2
     Error in pipeline configuration register.
     Expected: eeee, Received: rrrr.
```

Subphase 3 of Phase 1 Tests the signature of the handshaking lines between the ACP and the PLS cards. It takes a signature from Node 0 which is driven by the ACP/PLS handshaking lines while vectors are sent down the pipe to stimulate the pipeline. The FISAM test bit in the pipeline configuration register is asserted to break feedback paths to the clock stopper. SENDTORB and clipping are disabled.
If the diagnostic receives an unstable signature, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4650.3
Unstable signature received on root node
Node 0, Board name PLS, device number dddddddd.
Expected: xxxx, Received: unstable.
```

where 0 is the root node where the error is detected, xxxx is the known signature for root node 0, yyyy is the gathered signature, and dddddddd is the device number.
If the diagnostic receives a stable but incorrect signature from a root node, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4640.3
Signatures do not match the root node
Node 0, Board name PLS, device number dddddddd.
Expected: xxxx, Received: yyyy.
```

Subphase 4 of Phase 1 Tests the signature of the shadow pipeline. This takes a signature from Node 1 which is driven by the shadow pipelines while a simple vector list is sent down the pipe to stimulate the pipeline. The FISAM test bit in the pipeline configuration register is asserted to break feedback paths to the clock stopper. SENDTORB and clipping are disabled.

If the diagnostic receives an unstable signature from a root node, the following error message is displayed:

Bit_Pattern		*Hex_Value*
BitPattern[1]	=	X'FFFF'
BitPattern[2]	=	X'0000'
BitPattern[3]	=	X'5555'
BitPattern[4]	=	X'AAAA'
BitPattern[5]	=	X'FFFE'
BitPattern[6]	=	X'FFFD'
BitPattern[7]	=	X'FFFB'
BitPattern[8]	=	X'FFF7'
BitPattern[9]	=	X'FFEF'
BitPattern[10]	=	X'FFDF'
BitPattern[11]	=	X'FFBF'
BitPattern[12]	=	X'FF7F'
BitPattern[13]	=	X'FEFF'
BitPattern[14]	=	X'FDFF'
BitPattern[15]	=	X'FBFF'
BitPattern[16]	=	X'F7FF'
BitPattern[17]	=	X'EFFF'
BitPattern[18]	=	X'DFFF'
BitPattern[19]	=	X'BFFF'
BitPattern[20]	=	X'7FFF'
BitPattern[21]	=	X'0001'
BitPattern[22]	=	X'0002'
BitPattern[23]	=	X'0004'
BitPattern[24]	=	X'0008'
BitPattern[25]	=	X'0010'
BitPattern[26]	=	X'0020'
BitPattern[27]	=	X'0040'
BitPattern[28]	=	X'0080'
BitPattern[29]	=	X'0100'
BitPattern[30]	=	X'0200'
BitPattern[31]	=	X'0400'
BitPattern[32]	=	X'0800'
BitPattern[33]	=	X'1000'
BitPattern[34]	=	X'2000'
BitPattern[35]	=	X'4000'
BitPattern[36]	=	X'8000'

Table 4.2: MPLSD0B-1 Bit Patterns

```
***** MPLSDOB  ;1 - Phase 1  Error number 4650.4
Unstable signature received on root node
Node 1, Board name PLS, device number dddddddd.
Expected: xxxx, Received: unstable.
```

where 1 is the root node where the error is detected, xxxx is the known signature for the root node 1, yyyy is the gathered signature, and dddddddd is the device number.
If the diagnostic receives a stable but incorrect signature from a root node, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 1  Error number 4640.4
Signatures do not match the root node
Node 1, Board name PLS, device number dddddddd.
Expected: xxxx, Received: yyyy.
```

4.4.4.2 Phase 2

Phase 2 tests MULBUS paths to the PLS and FIFO. Phase 2 has three subphases.

Subphase 1 of Phase 2 Tests the upper 16-bit data path over the MULBUS to FIFO. A full set of test patterns (divided into X, Y, Z, and W vector groups) is used in this subphase. Each vector pattern is shifted to the upper 16-bits of the 32-bit MULBUS path and is then sent down the pipeline. The vector is read out of FIFO and is compared with the expected vector.
If there is an error in the upper 16-bit data path, the following error message is displayed:

```
***** MPLSDOB  ;2 - Phase 2  Error number 4605.1
Error in upper 16-bits of data to FIFO.
Section (X,Y,Z or W)
Expected: eeee, Received: rrrr.
```

where Section identifies which element of the vector is in error.

Subphase 2 of Phase 2 Tests the lower 16-bit data path over the MULBUS to FIFO. A full set of test patterns (divided into X, Y, Z, and W vector groups) is used in this subphase. Each vector pattern is shifted to the lower 16-bits of the 32-bit MULBUS path and is then sent down the pipeline. The vector is read out of FIFO and is compared with the expected vector.

If there is an error in the lower 16-bit data path, the following error message is displayed:

```
***** MPLSDOB  ;2 - Phase 2  Error number 4606.2
Error in lower 16-bits of data to FIFO.
Section (X,Y,Z or W)
Expected: eeee, Received: rrrr.
```

Subphase 3 of Phase 2 Tests the ability of the FIFO to hold multiple vectors. The test patterns are shifted to the upper 32 bits of the MULBUS and the entire vector list is loaded into FIFO. The list is checked to see if FIFO can contain all the vector patterns. If there is an error in FIFO, the following error message is displayed:

```
***** MPLSDOB  ;2 - Phase 2  Error number 4607.3
Error in FIFO.
Section (X,Y,Z or W)
Expected: eeee, Received: rrrr.
```

4.4.4.3 Phase 3

Phase 3 tests the Block Normalizer. A full set of patterns is shifted to bits 0 → 15, then to bits 1 → 16, then to bits 2 → 17, and so on until the patterns are shifted to bits 16 → 32. Each time the patterns are sent down the pipeline, the appropriate shifting must take place for the proper patterns to be returned from FIFO.

If an error is encountered in the shift test, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 3  Error number 4608.1
Error in shifter test.
Section (X,Y,Z or W)
Expected shift code = s
```

where "Section" identifies which element of the vector in which the error occurred. The expected shift code is the code that should be generated on NRMCODE[0-3].

If the diagnostic is reporting errors at this point, Xon or Xoff can be used to stop the execution of the diagnostic. Look at the values of NRMCODE[0-3] to identify the errors.

4.4.4.4 Phase 4

Phase 4 tests the perspective divider circuits. Phase 4 consists of two subphases.

Subphase 1 of Phase 4 Tests the basic data path through the perspective divider. It sets W to one (X'7FFF). Since X,Y,Z are divided by one (W), the results coming out of the divider should match the values going into the divider. The results are read through the DIV data register back to the ACP.
If an error is encountered in the perspective divide data path, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 4  Error number 4609.1
Error in perspective divide data.
Section (X/W,Y/W,Z/W or W/W)
Expected: xxxx, Received: rrrr.
```

Subphase 2 of Phase 4 Performs a variety of multiplications to test the perspective divider multiplier. It uses different values for W,X,Y,Z to test the multiplier.
If an error is encountered in the perspective divide multiplier, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 4  Error number 4610.2
Error in perspective divide multiplier.
Section (X/W,Y/W,Z/W or W/W)
Expected: xxxx, Received: rrrr.
```

4.4.4.5 Phase 5

Phase 5 tests the IYX data path through the viewport mapper. X and Y viewports and the W element of the vector are set to one (X'7FFF) so that data sent down the pipeline are multiplied by one, returning the same values from the IYX data register to the ACP.

If an error is encountered in the viewport mapper IYX data path, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 5  Error number 4611.1
IYX data error.
Vector count vvvv  Section (X or Y)
Expected: xxxx, Received: rrrr.
```

4.4.4.6 Phase 6

Phase 6 tests the viewport and intensity registers. Phase 6 writes a full set of patterns into the X and Y viewport and intensity registers. A vector with all elements equal to

one (7FFF) is sent down the pipeline so that the viewport/intensity value is multiplied is by one, giving the contents of the viewport/intensity register on the IYX bus as a result. Phase 6 has three subphases.

Subphase 1 of Phase 6 Tests the data path for the X viewport register. If an error is encountered in the X viewport register, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 6  Error number 4612.1
Error in viewport X.
Expected: xxxx, Received: rrrr.
```

Subphase 2 of Phase 6 Tests the data path for the Y viewport register. If an error is encountered in the Y viewport register, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 6  Error number 4613.2
Error in viewport Y.
Expected: xxxx, Received: rrrr.
```

Subphase 3 of Phase 6 Tests the intensity register.
If an error is encountered in the intensity register, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 6  Error number 4614.3
Error in intensity register.
Expected: xxxx, Received: rrrr.
```

4.4.4.7 Phase 7

Phase 7 tests viewport and intensity multipliers. For the intensity multiplier test, X and Y viewports are loaded with a value equal to one (7FFF). Phase 7 has two subphases.

Subphase 1 of Phase 7 Tests the intensity multiplier. The intensity register is loaded with one multiplication component, while the Z component of the vector contains the second multiplication component. The result is then read out on the IYX bus.
If an error is encountered in the intensity multiplier, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 7  Error number 4615.1
Error in the intensity multiplication.
Expected: xxxx, Received: rrrr.
```

Subphase 2 of Phase 7 Tests the viewport multiplier. One multiplication component is loaded into the X or Y viewport or intensity register. The second multiplication component is contained in the vector sent down the pipeline. The result is obtained from the IYX Bus.
If an error is encountered in the viewport multiplier, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 7  Error number 4616.2
Error in viewport multiplication.
Section (I,Y, or X)
Expected: xxxx, Received: rrrr.
```

4.4.4.8 Phase 8

Phase 8 tests the PLS clipping circuit. Each subphase tests clipping against a different clipping plane. Phase 8 sends a vector that requires clipping down the pipeline and detects if the clip occurred properly. It reads the clipped vector out of the FIFO to determine if the FIFO contains the correct vector. Subphases 1 through 4 cause the BADZ flag to be raised and determines if the BADZ flag can be detected. Phase 8 has six subphases.

Subphase 1 of Phase 8 Tests for a clip in positive X.
If a clipping flag for a clip in the positive X direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4622.1
CLIP flag not detected for a clip in positive X.
```

If a clipping flag for a clip in the positive X direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.1
Error in vector for clip in positive X.
X data error      X       Y       Z       W      Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW   IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW   IIII
-or-
Error in Bad Z detection.
```

Subphase 2 of Phase 8 Tests for a clip in negative X.
If a clipping flag for a clip in the negative X direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4623.2
CLIP flag not detected for a clip in negative X.
```

If a clipping flag for a clip in the negative X direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.2
Error in vector for clip in negative X.
X data error      X       Y       Z       W       Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
-or-
Error in Bad Z detection.
```

Subphase 3 of Phase 8 Tests for a clip in positive Y.
If a clipping flag for a clip in the positive Y direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4624.3
CLIP flag not detected for a clip in positive Y.
```

If a clipping flag for a clip in the positive Y direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.3
Error in vector for clip in positive Y.
X data error      X       Y       Z       W       Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
-or-
Error in Bad Z detection.
```

Subphase 4 of Phase 8 Tests for a clip in negative Y.
If a clipping flag for a clip in the negative Y direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4625.4
CLIP flag not detected for a clip in negative Y.
```

If a clipping flag for a clip in the negative Y direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.4
Error in vector for clip in negative Y.
X data error      X       Y       Z       W       Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
-or-
Error in Bad Z detection.
```

Subphase 5 of Phase 8 Tests for a clip in positive Z.
If a clipping flag for a clip in the positive Z direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4626.5
CLIP flag not detected for a clip in positive Z.
```

If a clipping flag for a clip in the positive Z direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.5
Error in vector for clip in positive Z.
Z data error      X       Y       Z       W       Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
```

Subphase 6 of Phase 8 Tests for a clip in negative Z.
If a clipping flag for a clip in the negative Z direction is not detected, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4627.6
CLIP flag not detected for a clip in negative Z.
```

If a clipping flag for a clip in the negative Z direction is detected but the vector read back is incorrect, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 8  Error number 4628.6
Error in vector for clip in negative Z.
Z data error      X       Y       Z       W       Int
Expected Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
Received Vector: XXXX    YYYY    ZZZZ    WWWW    IIII
```

4.4.4.9 Phase 9

Phase 9 tests the PLS signatures. Phase 9 tests all five PLS card signature nodes, nodes 0 through 4. It sets up the pipeline configuration register to enable clipping and transfer vectors to the refresh buffer. The viewport and intensity registers are set up and a list of vectors is sent down the pipeline. The viewport and intensity registers are changed and the vector list is again sent down the pipeline. This operation is performed on ten viewport settings. The modifiable frame sync option indicates how many times this operation is repeated.

If a Refresh Buffer card is present in the system, a signature is not taken on root node 4.

If an unstable signature error is encountered, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 9  Error number 4650.1
Unstable signature received on root node
Node rr, Board name PLS, device number dddddddd.
Expected: xxxx, Received: unstable.
```

If a stable but incorrect signature is encountered, the following error message is displayed:

```
***** MPLSDOB  ;1 - Phase 9  Error number 4640.1
Signatures do not match the root node
Node rr, Board name PLS, device number dddddddd.
Expected: xxxx, Received: yyyy.
```

4.4.4.10 Phase 10

Phase 10 tests the PLS/Refresh Buffer card interface. A full set of patterns is transferred from the PLS card to the RFB card, first selecting the downer memory buffer and then selecting the upper memory buffer. Subphases 1 and 2 load vectors directly through the IYX data register to the refresh buffer. Subphases 3 and 4 send vectors down the pipeline to the refresh buffer. The patterns are then read out of RFB memory by the GCP and are compared with the expected data. If there is no Refresh Buffer card in the system, the diagnostic reports this condition. Phase 10 has four subphases.

Subphase 1 of Phase 10 Performs an IYX data test on the interface to the downer memory buffer.

If an error is encountered in the downer memory buffer, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 10  Error number 4629.1
Data error in PLS/Refresh Buffer Interface.
Expected: xxxx, Received: rrrr.
```

Subphase 2 of Phase 10 Performs an IYX data test on the interface to the upper memory buffer.
If an error is encountered in the upper memory buffer, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 10  Error number 4629.2
Data error in PLS/Refresh Buffer Interface.
Expected: xxxx, Received: rrrr.
```

Subphase 3 of Phase 10 Performs a vector data test on the interface to the downer memory buffer.
If an error is encountered in the downer memory buffer, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 10  Error number 4630.3
Vector data error in PLS/Refresh Buffer Interface.
Expected: xxxx, Received: rrrr.
```

Subphase 4 of Phase 10 Performs a vector data test on the interface to the upper memory buffer.
If an error is encountered in the upper memory buffer, the following error message is displayed.

```
***** MPLSDOB  ;1 - Phase 10  Error number 4630.4
Vector data error in PLS/Refresh Buffer Interface.
Expected: xxxx, Received: rrrr.
```

4.4.4.11 Phase 11

Phase 11 is an optional phase that implements the PLS debugger. This phase requires a functional knowledge of the PS 350 PLS card. Phase 11 is used instead of loop-on-error to repeatedly send a vector or vectors down the pipeline. The pipeline configuration register and the viewport registers must be set up correctly, as explained below. To simulate a loop-on-error for any previously failing phase, use the set-up information provided in the failing phase.

The PLS debugger displays the following menu of commands:

```
Available Commands:
```

```
Examine
Load
Run
Clear Matrix Memory
Initialize Refresh Buffer
Show present configuration
Quit debug phase
```

Enter the commands as listed above. After the first few characters are typed, the debugger recognizes the command and finishes printing out the command. If an invalid command or a misspelled command is entered, the debugger will return "Invalid Command."

The Examine, Load, and Run commands have options. To see a list of options, enter the command, then enter HELP, and a list of options will be displayed.

4.4.4.11.1 Examine The Examine command allows reading of specified registers and memory locations. The Examine HELP command prints out the options:

```
Available Examine Options:
  STATUS
  MATRIX MEMORY
  PIPELINE CONFIGURATION REGISTER
  REFRESH BUFFER MEMORY
```

Status The Examine STATUS option reads and displays PLS status information. Only the upper four bits are valid:

*MULBLO	Bit 12	If low, the Lower MULBUS cable is connected to the ACP card.
MULBHI	Bit 13	If low, the Upper MULBUS cable is connected to the ACP card.
RBEXISTS	Bit 14	If low, the Refresh Buffer card is in the system.
FIFOEMPTY	Bit 15	If low, the FIFO is empty.

Matrix Memory The Examine MATRIX MEMORY option displays the contents of matrix memory. It prompts for the matrix memory address to be read. It prompts for how many consecutive locations are to be read. It prints out results in X,Y,Z,W format.

Pipeline Configuration Register The Examine PIPELINE CONFIGURATION REGISTER option displays the label on each bit in the register. The option then prints out the contents of the pipeline configuration register.

Refresh Buffer Memory The Examine REFRESH BUFFER MEMORY option displays the contents of upper or lower refresh buffer memory. It prompts to select upper or lower memory. It then prompts for a start address and how many locations are to be read. The option finally displays results for I,Y,X, and V.

4.4.4.11.2 Load The Load Command allows loading a register or memory location. The Load HELP command prints out the options:

```
Available Load Options
  MATRIX MEMORY
  PIPELINE CONFIGURATION REGISTER
  DEFAULT VECTOR LIST
  REFRESH BUFFER COMMAND
  REFRESH BUFFER MEMORY
  TEST PATTERNS
  VECTOR LIST
  VIEWPORT REGISTERS
```

Matrix Memory The Load MATRIX MEMORY option loads a specified value into matrix memory. It prompts for an address and prints out the current contents of that address. The option then prompts for the data to be loaded into that address.

Pipeline Configuration Register The Load PIPELINE CONFIGURATION REGISTER option loads a value into the pipeline configuration registers. It prints out the bit labels for the register and the current contents of the register. It then prompts for a value to be loaded into the register.

Default Vector List The Load DEFAULT VECTOR LIST option loads a default list of vectors into matrix memory that is used in the Run commands.

Refresh Buffer Command The Load REFRESH BUFFER COMMAND option prompts for a command and sends it to the Refresh Buffer card output processor.

Refresh Buffer Memory The Load REFRESH BUFFER MEMORY option prompts to select upper or lower memory. It then prompts for a start address and how many locations are to be loaded. It prompts for I,Y,X and V values to be loaded into memory.

Test Patterns The Load TEST PATTERNS option loads default test patterns into matrix memory. The patterns are used by the refresh buffer interface test.

Vector List The Load VECTOR LIST option prompts for how many vectors will be loaded. It prompts for each X,Y,Z,W vector and loads them into matrix memory. The Run commands use these values.

Viewport Registers The Load VIEWPORT REGISTERS option prompts for X,Y,Z viewport and intensity register values and loads them into the registers. The Run command uses these values.

4.4.4.11.3 Run The Run Commands run one of the tests listed below. Each test, with the exception of the RFB interface test, sends the vector list specified with the Load command down the pipeline. The viewport and pipeline configuration registers should also be loaded. Bit 0 of the intensity register is used as the Draw/*Move bit. The Run HELP command prints out the options.

```
Available Run Test Options:
  FIFO TEST
  IYX TEST
  PERSPECTIVE DIVISION TEST
  CLIP TEST
  VECTOR LOOP
  REFRESH BUFFER INTERFACE
  SIGNATURE TEST
```

Fifo Test The Run FIFO TEST option sends the specified vector list down the pipeline. The vectors and intensity data are read out of the FIFOs and are displayed. For this test function properly, the KEEPALL bit in the pipeline status register must be set.

IYX Test The Run IYX TEST option sends the specified vector list down the pipeline. These vectors are read out of the IYX data register and are displayed as I,Y, and X. At this point, the vectors are divided by W, clipped, and multiplied by the viewport registers.

Bit 15 of the IYX data register is data valid. Bit 14 is Draw/*Move. Bit 13 is Show Endpoint.

Perspective Division Test The Run PERSPECTIVE DIVISION TEST option sends the specified vector list down the pipeline. The vectors are read out of the DIV data register and are the result of a division by the W component of the vector. Bit 0 of the DIV data register is the most significant of the two sign bits from the result of the perspective division and is set if a clip occurs.

Clip Test The Run CLIP TEST option sends the specified vector list down the pipeline while checking for a clip. If a clip is detected, the vector value is read out of the FIFOs. Bad Zs are flagged. The clipped vector is the second vector printed out, unless the first vector sent down the pipeline was clipped. The pipeline configuration register must be set to enable clipping. If the IGNOREZCLIP bit in the pipeline configuration register is set, the BADZ flag can also be detected.

Vector Loop The Run VECTOR LOOP option sends the specified vector list down the pipeline continuously. This option can be used to simulate a loop on error.

Refresh Buffer Interface The Run REFRESH BUFFER INTERFACE option prompts to select upper or lower RFB memory. It then sends the test pattern loaded by LOAD TEST PATTERNS to the selected memory buffer starting at location X'0008. Use EXAMINE REFRESH BUFFER MEMORY to see if the patterns were written into memory correctly. It does not send vectors down the pipe. It writes the pattern directly into the RFB interface register.

Signature Test The Run SIGNATURE TEST option displays the contents of the signature for the specified node. It prompts for the signature node. It sends vectors down the pipe and then reads and reports the signature gathered.

4.4.4.11.4 Clear Matrix Memory The Clear Matrix Memory command clears all of matrix memory.

4.4.4.11.5 Initialize Refresh Buffer The Initialize Refresh Buffer command initializes the Refresh Buffer card. This must be done before vectors or data can be accepted by the Refresh Buffer.

4.4.4.11.6 Show Present Configuration The Show Present Configuration command shows the contents of the pipeline configuration register, the viewport registers, and the vector list loaded by the Load command.

4.4.4.11.7 Quit Debug Phase The Quit Debug Phase exits Phase 11.

4.4.5 Error Analysis

Tables 4.3 and 4.4 provide the error numbers and corresponding error messages for the Pipeline Subsystem Analytic Diagnostic.

MPLSD0B Error Messages	
4601.1	Lower MULBUS cable not connected correctly
4602.1	Upper MULBUS cable not connected correctly
4603.1	FIFO should be empty, but it is not
4604.2	Error in pipeline configuration register. Expected: eeee, Received: rrrr.
4605.1	Error in upper 16-bits of data to FIFO. Section (X, Y, Z or W) Expected: eeee, Received: rrrr.
4606.2	Error in lower 16-bits of data to FIFO. Section (X, Y, Z or W) Expected: eeee, Received: rrrr.
4607.3	Error in FIFO. Section (X, Y, Z or W) Expected: eeee, Received: rrrr.
4608.1	Error in shifter test. Section (X, Y, Z or W) Expected shift code = s
4609.1	Error in perspective divide data. Section (X/W, Y/W, Z/W or W/W) Expected: xxxx, Received: rrrr.
4610.2	Error in Perspective Divide Multiplier. Section (X/W, Y/W, Z/W or W/W) Expected: xxxx, Received: rrrr.
4611.1	IYX data error. Vector count vvvv Section (X or Y) Expected: xxxx, Received: rrrr.
4612.1	Error in viewport X. Expected: xxxx, Received: rrrr.
4613.2	Error in viewport Y. Expected: xxxx, Received: rrrr.
4614.3	Error in intensity register. Expected: xxxx, Received: rrrr.
4615.1	Error in the intensity multiplication. Expected: xxxx, Received: rrrr.
4616.2	Error in viewport multiplication. Section (I, Y, or X) Expected: xxxx, Received: rrrr.
4622.1	CLIP flag not detected for a clip in positive X.
4623.2	CLIP flag not detected for a clip in negative X.
4624.3	CLIP flag not detected for a clip in positive Y.
4625.4	CLIP flag not detected for a clip in negative Y.
4626.5	CLIP flag not detected for a clip in positive Z.
4627.6	CLIP flag not detected for a clip in negative Z.

Table 4.3: MPLSD0B Error Messages Part One

MPLSD0B Error Messages	
4628.1	Error in vector for clip in positive X. -or- Error in Bad Z detection.
4628.2	Error in vector for clip in negative X. -or- Error in Bad Z detection.
4628.3	Error in vector for clip in positive Y. -or- Error in Bad Z detection.
4628.4	Error in vector for clip in negative Y. -or- Error in Bad Z detection.
4628.5	Error in vector for clip in positive Z.
4628.6	Error in vector for clip in negative Z.
4629.1	Data error in PLS/Refresh Buffer Interface. Expected: xxxx, Received: rrrr.
4629.2	Data error in PLS/Refresh Buffer Interface. Expected: xxxx, Received: rrrr.
4630.1	Unstable signature received for root node rr, device number ddddddddd. Expected: xxxx, Received: unstable.
4630.3	Vector data error in PLS/Refresh Buffer Interface. Expected: xxxx, Received: rrrr.
4630.4	Vector data error in PLS/Refresh Buffer Interface. Expected: xxxx, Received: rrrr.
4640.1	Signatures do not match on root node rr, device number ddddddddd. Expected: xxxx, Received: yyyy.
4640.3	Signatures do not match the root node Node 0, Board name PLS, device number ddddddddd. Expected: xxxx, Received: yyyy.
4640.4	Signatures do not match the root node Node 1, Board name PLS, device number ddddddddd. Expected: xxxx, Received: yyyy.
4650.3	Unstable signature received on root node Node 0, Board name PLS, device number ddddddddd. Expected: xxxx, Received: unstable.
4650.4	Unstable signature received on root node Node 0, Board name PLS, device number ddddddddd. Expected: xxxx, Received: unstable.

Table 4.4: MPLSD0B Error Messages Part Two

4.5 Low Cost Peripherals Function Buttons Diagnostic

The Function Buttons unit is a programmable interactive device that is controlled by an internal microprocessor. It connects to the MUX or Multiplexer in the same fashion as the other interactive devices, communicating bi-directionally and asynchronously at 2400 baud through a serial port. There are 32 un-labeled buttons on the face of the cabinet, each of which lights to show an "on" condition. The buttons are numbered left to right and top to bottom in the following pattern:

	0	1	2	3	
4	5	6	7	8	9
10	11	12	13	14	15
16	17	18	19	20	21
22	23	24	25	26	27
	28	29	30	31	

The Function Buttons Unit for the New Peripheral Set is connected to the front of the MUX box underneath the Buttons label(which is the second connector from the left). The PS 300 Style Function Button Set is usually connected to Port # C on the Data Concentrator.

When the Function Buttons Unit is connected to a Data Concentrator, the JCP uses the printable ASCII characters X'40' through X'5F' to turn on a light for a specific button. Similarly, the JCP uses the printable ASCII characters X'60' through X'7F' to turn off a specific light. The ASCII codes are assigned as shown in Table 4.5.

4.5.1 Light Control

When used with the Low Cost Peripheral Set, the Function Button unit controls the lights differently.

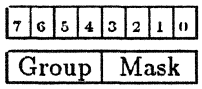
For the purpose of turning the lights of the Buttons box on or off, the lights are logically grouped into eight groups of four lights each. The lights of the box are then turned on and of respectively by sending a message consisting of one to eight bytes to it. The four more-significant bits of each byte contains the identification number for a four-light group; the four less-significant bits contain a mask which turn on (if the corresponding bit is set) or off (if the bit is clear) the light. This is shown in Figure 4.2 where the Group Number is binary 0000 thru 0111 and Light Mask 1's and 0's turn lights on and off.

The Function Button Light Groups are defined in Table 4.6.

Button #	*On*	*Off*	*Button #*	*On*	*Off*	*Button #*	*On*	*Off*
0	@	`	11	K	k	22	V	v
1	A	a	12	L	l	23	W	w
2	B	b	13	M	m	24	X	x
3	C	c	14	N	n	25	Y	y
4	D	d	15	O	o	26	Z	z
5	E	e	16	P	p	27	[	{
6	F	f	17	Q	q	28	}	—
7	G	g	18	R	r	29	]	}
8	H	h	19	S	s	30	↑	~
9	I	i	20	T	t	31	←	DEL
10	J	j	21	U	u			

Table 4.5: Function Button Toggle Codes

Figure 4.2: Function Button Light Control Message Byte



Any byte or combination of bytes may be sent in a message, depending on which of the lights must be turned on or turned off. Turning all lights on, turning all lights off or changing the state of at least one byte of each of the eight groups would require an eight-byte message to be sent. Changing the state of one to four lights in a single four-light group would require only a one-byte message to be sent.

4.5.2 Reporting Selections

The Buttons box reports that a key has been depressed simply by sending a single byte to the Joint Control Processor. The value of the byte is given by adding the hexadecimal value of the key number to the hexadecimal value x'3F'. Thus the first sixteen keys are numbered x'40' to x'4F' and the second group of sixteen keys are numbered x'50' to

Table 4.6: Function Button Light Groups

Group Number	*Description*
b'0000'	Group for lights 1 → 4
b'0001'	Group for lights 5 → 8
b'0010'	Group for lights 9 → 12
b'0011'	Group for lights 13 → 16
b'0100'	Group for lights 17 → 20
b'0101'	Group for lights 21 → 24
b'0110'	Group for lights 25 → 28
b'0111'	Group for lights 29 → 32

x'5F'. Only one message per keystroke is reported.

4.5.3 Functional Description

The Function Buttons diagnostic consists of six phases (Phase 5 is applicable only when the Function Buttons Unit is connected to the Data Concentrator):

Phase 1 Determines if the Function Buttons unit is properly connected to the system and if the Function Buttons unit can respond to the system's inquire message.

Phase 2 Determines if each button's light can be separately controlled.

Phase 3 Determines if all buttons work. The user must depress each button separately to tell if it is operating correctly.

Phase 4 Determines if each button can control its light.

Phase 5 Determines that the Function Buttons unit can accurately report back its light status to the JCP. The diagnostic does this by sending a bit pattern to the Function Buttons unit and then requesting a report of the same bit pattern back. The report is then compared to the original pattern to ensure that both are identical.

Phase 6 Invokes the Function Buttons confidence test.

4.5.4 Parameter Modifications

Note: This section allows the user to modify parameters for the Function Buttons Diagnostic. If the Function Buttons are connected to the MUX Box the diagnostic will still ask for responses during "M"odify, however it will ignore whatever is typed.

Two test variables for the function buttons diagnostic may be modified using the "M"odify command before the diagnostic begins execution. The modifiable variables are:

(a) The Communications Connector Panel Port Number (2-5)
(b) The Data Concentrator Port Indicator (A-F)

Should one or more of these parameters be modified illegally, an information message is output to the diagnostic terminal. The possible messages are:

```
Communications connector panel port is out of range.
Communications connector panel port is already in use.
Data concentrator port is out of range.
Data concentrator port is already in use.
Data concentrator port timed out.
Data concentrator illegal initialization response.
```

The Communications Connector Panel port parameter specifies which Communications Connector Panel port is to be used. The default is Port 5. This variable may be changed when the following prompt appears:

```
Enter Communications Connector Panel port number (2-5).
```

The Data Concentrator parameter specifies whether or not the Communications Connector panel is connected to a Data Concentrator. By default, the connection is assumed to be made. After the following prompt appears, this variable may be modified:

```
Is Communications Connector Panel port connected to the Data
Concentrator?
```

If the user does not answer with an upper or lower case "Y," the answer is assumed to be no. A negative response prevents the Data Concentrator prompt (described next) from appearing.

The Data Concentrator port parameter indicates which Data Concentrator port is to be used for the Function Buttons. The default is Port D. This variable may be changed when the following prompt appears:

```
Enter Function Buttons Data Concentrator port letter (A-F).
```

To exit the "M"odify, type "y", "Y", or "yes" to the following prompt:

```
Do you want to exit Modify?
```

4.5.5 Detailed Phase Description

4.5.5.1 Phase 1

This first phase of the diagnostic determines if the Function Buttons unit is correctly connected to the system, and determines if the Function Buttons unit can respond to the PS 300/390 inquire message (ASCII ENQ, CTRL E, or X'05'). If the Function Buttons Box is connected to a Data Concentrator, the Diagnostic send out a X'80' and waits for .5 seconds. If no response is received in that time, the following error message is displayed. If the Function Buttons unit is connected to a MUX Box, the system sends out the CTRL E to the Function Buttons unit and then waits .5 seconds. If no response occurs within that period, the following message is produced:

```
***** BTNDOB  ;1 - Phase 1 Error number 3507.1
Function Box is not responding to inquiry.
```

An invalid response generates the following error message:

```
***** BTNDOB  ;1 - Phase 1 Error number 3508.1
Invalid inquiry response message
Expected:  BTNxxx Received eeeeee
```

4.5.5.2 Phase 2

The second phase of the diagnostic is used to ensure that each button light can be separately controlled. The diagnostic does this by transmitting the ASCII characters shown in Table 4.5 to the Functions Button unit at 300 ms intervals. This phase of the diagnostic consists of five parts that illuminates each light or turns off the lights for all or some buttons. The phase stops after each part is completed so that the user has time to verify that the correct buttons are lit. No error messages are generated by this phase.

Subphase 1 Turns on all button lights simultaneously.

Subphase 2 Turns off all button lights simultaneously.

Subphase 3 Turns each button light on and back off, one at a time. The buttons are illuminated and turned off in order, starting with button 0 and ending with button 31.

Subphase 4 Starts with button 0 and turns on each button individually until all of them through button 31 are illuminated.

Subphase 5 Turns off all the button lights that were lit during Subphase 4. Again, the diagnostic proceeds from button 0 to button 31, turning off each light.

4.5.5.3 Phase 3

The third phase of the diagnostic is used to determine if each of the 32 buttons can report to the system. The user must depress each button separately and ensure that the button's number is displayed on the diagnostic terminal. This phase does not end until the user terminates it by striking any key on the diagnostic terminal. If the system receives an invalid character from the Function Buttons unit, the following error message is produced:

```
***** BTNDOB  ;1 - Phase 3 Error number 3509.1
Invalid character code  :  X'hh'
```

4.5.5.4 Phase 4

The fourth diagnostic phase provides further tests to ensure that each button works correctly. This time the user consecutively depresses any number of individual buttons. Each time a button is pressed, its light should toggle on or off. Also, the number of the button being toggled appears on the diagnostic terminal. This phase of the diagnostic continues until the user presses any key on the diagnostic terminal. If an invalid character is received by the system as a button is depressed, the following error message appears:

```
***** BTNDOB  ;1 - Phase 4 Error number 3509.1
Invalid character code   : X'hh'
```

4.5.5.5 Phase 5

Note: Phase 5 only runs on the standard Function Buttons Box. It does not run on the Low Cost Function Buttons.

The fifth diagnostic phase is a measure of the Function Buttons' ability to accurately report to the system if any button light is on or off. The diagnostic tests this by writing 20 different combinations of light and unlight buttons (i.e, 20 different 32-bit strings) to the Function Buttons unit and asking the Function Buttons unit to report back which buttons are lit and which buttons are unlight for each of the 20 strings. The twenty different patterns are as follows:

Pattern				
Bit 0			Bit 31	Index
01010101	01010101	01010101	01010101	0

00110011	00110011	00110011	00110011	1
00001111	00001111	00001111	00001111	2
00000000	11111111	00000000	11111111	3
00000000	00000000	11111111	11111111	4
00000000	00000000	00000000	00000000	5
10101010	10101010	10101010	10101010	6
11001100	11001100	11001100	11001100	7
11110000	11110000	11110000	11110000	8
11111111	00000000	11111111	00000000	9
11111111	11111111	00000000	00000000	10
11111111	11111111	11111111	11111111	11
10000000	10000000	10000000	10000000	12
01000000	01000000	01000000	01000000	13
00100000	00100000	00100000	00100000	14
00010000	00010000	00010000	00010000	15
00001000	00001000	00001000	00001000	16
00000100	00000100	00000100	00000100	17
00000010	00000010	00000010	00000010	18
00000001	00000001	00000001	00000001	19

If the pattern reported by the Function Buttons unit does not match the one originally transmitted to it by the diagnostic, the following error message is displayed on the diagnostic terminal:

```
***** BTNDOB  ;1 - Phase 5 Error number 3510.x
Response error.  Expected:  xxxxxxxx   Received:  yyyyyyyy
```

The index number shown in the pattern table is the "x" portion of the Error number. The index number identifies which of the twenty 32-bit combinations was erroneously reported by the Function Buttons unit. Once the index number is determined, the actual bit in error may be identified by comparing the eight hex digits in the Expected and Received values. For example, the actual error message may appear as follows:

```
***** BTNDOB  ;1 - Phase 5 Error number 3510.2
Response error.  Expected:  F0F0F0F0   Received:  E0F0F0F0
```

From this information, it may first be determined that the string identified by Index Number 2 was received from the system by the Function Buttons unit and then erroneously reported back to the system. The eight hex values in the Expected and Received

values must then be compared to determine the bit in error. The first hex character in both values represents bits (and buttons) 31, 30, 29 and 28; the second hex character represents the next four button lights (27, 26, 25, and 24), and so on. In the example given, the first received hex character is erroneous. By comparing the received value (E) to the expected value (F) it may be readily determined that bit 31 is wrong.

If the Function Buttons unit does not respond at all to the status request, the following error message is reported:

```
***** BTNDOB  ;1 - Phase 5 Error number 3511.1
Timeout error.
```

4.5.5.6 Phase 6

This final phase of the diagnostic invokes the Function Button confidence test routines to test internal RAM and ROM. First, the test command (CTRL T) is sent to the Function Buttons unit to start testing. The system displays the following ASCII characters on the diagnostic terminal as the confidence tests are completed:

```
"A" indicates that the confidence tests have started.
"B" indicates that the internal RAM test has completed.
"C" indicates that the ROM test has completed.
"D" indicates that the external RAM test has completed.
```

If an error is detected while receiving the start test character, the following message is produced:

```
***** BTNDOB  ;1 - Phase 6 Error number 3512.1
Bad initial response from confidence tests.
```

If the Function Buttons unit does not respond at all to the CTRL T, the following message appears:

```
***** BTNDOB  ;1 - Phase 6 Error number 3522.2
No initial response from confidence tests.
```

If an error is detected during the internal RAM test, the following message appears:

```
***** BTNDOB  ;1 - Phase 6 Error number 3513.1
MC6803 internal memory bad.
```

If an error is detected during the internal ROM test, the following message appears:

```
***** BTNDOB  ;1 - Phase 6 Error number 3514.1
EPROM checksum invalid.
```

If an error is detected during the external RAM test, the following message is produced:

```
***** BTNDOB  ;1 - Phase 6 Error number 3515.1
MC6803 external memory bad.
```

4.5.6 Error Analysis

The error messages produced during the Function Buttons diagnostic have the following meanings:

3501	Communications Connector Panel port is out of range.
3502	Communications Connector Panel port is already in use.
3503	Data Concentrator port is out of range.
3504	Data Concentrator port is already in use.
3505	Data Concentrator timed out.
3506	Data Concentrator illegal initialization response.
3507	Function Buttons unit is not responding to inquiry.
3508	Invalid inquiry response message. Expected: BTNxxx Received eeeeee
3509	Invalid character code.
3510	Bad response in reporting the buttons' light status.
3511	Timeout error.
3512	No initial response from confidence tests.
3513	MC6803 Internal Memory Error.
3514	EPROM Checksum invalid.
3515	MC6803 External Memory Bad.
3516	No indication of test completion.
3553	Multiplexer Box Point is out of range.
3554	Multiplexer Box Point is already in use.
3557	Multiplexer Box Timed Out.
3558	Multiplexer Box illegal initiation response.
3559	MUX failed self test.

Table 4.7: Function Button Error Messages

Notes On Preliminary Diagnostics For The PS 390

Date: 5 May 87
Version: NC.01

Diagnostic Disk G

This disk contains the following diagnostics and their associated microcode files:

RBED0A Bit Slice Processor diagnostic Including the YASD debugger RBED0A.DAT (μcode)

RBED1A Graphics Pipeline Diagnostic RBED1A.DAT (μcode)

RBED2A Pixel Processor and Frame Buffer diagnostic including a Visual Debugger called VISBUG FBTST.DAT (μcode)

MPLSD0A Phase 10 tests the Input FIFO on the RBE card. MPLSIG.TXT(signature file) PS390.DAT (μcode)

Diagnostic Disk H

This is just a temporary disk name. The new peripheral diagnostics are on this disk. The peripheral Mux box required some changes in the diagnostic operating system (A9.V01). When this new operating system is tested and released for all the diagnostic disks then these peripheral diagnostics will be released on diagnostic disk B.

BTND0B Function Button diagnostic for old and new buttons. Works on both old and new peripheral Mux boxes and it will work on a data concentrator.

CDLD0B Control Dial diagnostic. Works on both old and new style of control dials.

TABD1C GTCO Tablet diagnostic. Works with both the old and new peripheral Mux box and the data concentrator.

ECPD0A Energy Card diagnostic.

Note: YOU ARE NOW TESTING A NEW VERSION OF DIAGOS (A9.V01) with these preliminary diagnostics. Report any bugs by mail to BTURNER (CESS) on the CM780.

The new diagnostics fix the data concentrator problems in the A8 release. All the peripherals and the F15 will work on a GCP system if you boot this operating system before running them.

It automatically determines the configuration of the system. PS300, PS350 or PS390. (JCP or GCP)

It initializes the new peripheral multiplexer boxes. If you have a tablet connected with the stylus on the tablet you will have trouble booting diagnostics. Remove the stylus from the tablet.

Appendix A

PAL Definitions for the PS 390

A.1 PLS Transfer State Machine

Table A.1: PLS Transfer State Machine Input/Output Pin Signal Descriptions

Pin #	*Signal*	*Input Pin Descriptions*
Pin 2	*resetb	Active low, used to reset state machines.
Pin 3	plsready	Active high, synchronized version of plsready from the pls, indicates the the pls has data ready for bit-slice.
Pin 4	*fifofull	Active low, indicates that the FIFO is full.
		Output Pin Descriptions
Pin 19	*rbdone	Active low, indicates to PLS that the transfer of data has completed.
Pin 18	*enable_x	Active low, selects the X word from the PLS buffer.
Pin 17	*enable_y	Active low, selects the Y word from the PLS buffer.
Pin 16	*enable_i	Active low, selects the X word from the PLS buffer.
Pin 15	*enable_v	Active low, selects the I word from the PLS buffer, also resets the plsready flip_flop.
Pin 14	*writefifo	Active low, write signal for the fifo.

Table A.2: PLS Transfer State Machine State Descriptions

Current State	=	*enable_x Nrben_x	*enable_y Nrben_y	*enable_i Nrben_i	*enable_v Nrben_v	*writefifo NWriteFifo	*rbdone Nrbdone
Idle	=	1	1	1	1	1	1
TransI	=	1	1	0	1	1	1
WrFifoI	=	1	1	0	1	0	1
TransY	=	1	0	1	1	1	1
WrFifoY	=	1	0	1	1	0	1
TransX	=	0	1	1	1	1	1
WrFifoX	=	0	1	1	1	0	1
SendRBdone	=	1	1	1	1	1	0
ClearReadyFlag	=	1	1	1	0	1	1
TransferMode	=	PlsReady	NReset1	NFifoFull			
PLSandFifoReady	=	1	1	1			
PLSnotReady	=	0	1	X			
FifoFull	=	1	1	0			
Reset	=	X	0	X			

```
State Diagram TransferState

Idle: if (TransferMode == PLSandFifoReady) then TransI else Idle;

TransI:
          case (TransferMode == PLSandFifoReady) : WrFifoI;
               (TransferMode == FifoFull)        : TransI;
               (TransferMode == Reset)           : Idle;
          endcase;

WrFifoI:  if (TransferMode == FifoFull) then WrFifoI else TransY;

TransY:
          case (TransferMode == PLSandFifoReady) : WrFifoY;
               (TransferMode == FifoFull)        : TransY;
               (TransferMode == Reset)           : Idle;
          endcase;

WrFifoY:  if (TransferMode == FifoFull) then WrFifoY else TransX;

TransX:
          case (TransferMode == PLSandFifoReady) : WrFifoX;
               (TransferMode == FifoFull)        : TransX;
               (TransferMode == Reset)           : Idle;
          endcase;

WrFifoX:  if (TransferMode == FifoFull) then WrFifoX else SendRBdone;

SendRBdone:       goto ClearReadyFlag;

ClearReadyFlag:   goto Idle;
```

Table A.3: Fifo to Fifo Buffer State Machine Input/Output Pin Signal Descriptions

Pin #	*Signal*	*Input Pin Descriptions*
Pin 5	*resetb	Active low, see Pin 2 Table A.1.
Pin 6	*fifoempty	Active low, indicates the the FIFO is empty.
Pin 7	*sdfifo	Active low, indicates that the bitslice read.
		Output Pin Descriptions
Pin 13	*readfifo	Active low, read signal for the FIFO.
Pin 12	fifobufready	Active high, indicates to the bitslice that there is data ready in the FIFO.

InputFifoCntrl	=	NReset1	NFifoEmpty	NRFifo
Reset1	=	0	X	X
FifoHasData	=	1	1	1
TransferAgain	=	1	1	0
HmsReadsBuf	=	1	0	0
Noop	=	1	0	1

FifoBufState	=	NReadFifo	FifoBufReady
FifoBufIdle	=	1	0
TransferToBuf	=	0	1
Wait	=	1	1
UndefState0	=	0	0

State Diagram for FIFO to FifoBuffer Controller

State Diagram FifoBufState

```
FifoBufIdle:
              case (InputFifoCntrl == Reset1) : FifoBufIdle;
                   (InputFifoCntrl == FifoHasData) : TransferToBuf;
                   (InputFifoCntrl == HmsReadsBuf) : FifoBufIdle;
                   (InputFifoCntrl == TransferAgain) : TransferToBuf;
                   (InputFifoCntrl == Noop) : FifoBufIdle;
```

```
                endcase;

    TransferToBuf:
                case (InputFifoCntrl == Reset1) : FifoBufIdle;
                     (InputFifoCntrl == FifoHasData) : Wait;
                     (InputFifoCntrl == HmsReadsBuf) : Wait;
                     (InputFifoCntrl == TransferAgain) : Wait;
                     (InputFifoCntrl == Noop) : Wait;
                endcase;

    Wait:
                case (InputFifoCntrl == Reset1) : FifoBufIdle;
                     (InputFifoCntrl == FifoHasData) : Wait;
                     (InputFifoCntrl == HmsReadsBuf) : FifoBufIdle;
                     (InputFifoCntrl == TransferAgain) : TransferToBuf;
                     (InputFifoCntrl == Noop) : Wait;
                endcase;

    UndefState0:  GOTO FifoBufIdle;
```

Explanation of symbols:

1	= logic High
0	= logic Low
X	= don't care

A.2 Input Fifostack Bus Controller State Machine

Table A.4: Input FSBC Controller

Signal	*Pin #*	*Input FSBC Controller Inputs*
clk	Pin 1	
NReset	Pin 2	Reset State Machine
NInFsbcPreq	Pin 3	Fsbc is ready for input
HmsInPack	Pin 4	Hms is done writing Fsbc
NDyInFsbc	Pin 5	Hms writes Fsbc buffer register
FsbcReg0	Pin 6	Fsbc Register address bit 0
FsbcReg1	Pin 7	Fsbc Register address bit 1
FsbcReg2	Pin 8	Fsbc Register address bit 2
NClrLowWord	Pin 9	Clr Fsbc Low word before input
chip ground	Pin 10	
chip enable	Pin 11	
Input FSBC Controller Output Latches		
NFsbcReady	Pin 12	Ready signal for Hms
NFsbcWrite	Pin 13	InFsbc write strobe
NFsbcPack	Pin 14	Packet acknowledge for InFsbc
InFsbcAdr0	Pin 15	Fsbc Reg address
InFsbcAdr1	Pin 16	Fsbc Reg address
InFsbcAdr2	Pin 17	Fsbc Reg address
NClrLsw	Pin 18	Buffer has data for MBS
State0	Pin 19	State bit for controller
chip vcc	Pin 20	

FsbcControlState	=	State0	NFsbcReady	NFsbcWrite	NFsbcPack	
FsbcIdle	=	0	1	1	1	The IDLE state
FsbcWrState	=	0	0	0	1	Write a register in the FSBC
ReadyState	=	0	0	1	1	Let the HMS know that the FSBC is ready
FsbcPackState	=	0	0	1	0	Send PACK to the InputFsbc
FsbcWrPackState	=	1	0	0	1	Write a register in the FSBC and this state always followed by packstate
FsbcUndefState4	=	0	1	0	0	
FsbcUndefState5	=	0	1	0	1	
FsbcUndefState6	=	0	1	1	0	
FsbcUndefState8	=	1	0	0	0	
FsbcUndefStateA	=	1	0	1	0	
FsbcUndefStateB	=	1	0	1	1	
FsbcUndefStateC	=	1	1	0	0	
FsbcUndefStateD	=	1	1	0	1	
FsbcUndefStateE	=	1	1	1	0	
FsbcUndefStateF	=	1	1	1	1	

Table A.5: Input Fsbc Controller State Descriptions

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

Fsbc Controller Input Signal Modes

InputMode	=	NReset	NInFsbcPreq	HmsInPack	NDyInFsbc
Reset	=	0	X	X	X
FsbcIsReady	=	1	0	0	1
HmsWrBuf	=	1	0	0	0
HmsWrPack	=	1	0	1	1
HmsWrPackBuf	=	1	0	1	0
DontCare	=	1	X	X	X

Input Vector for the Fsbc Register Address

InputRegVec	=	NDyInFsbc	FsbcReg2	FsbcReg1	FsbcReg0
NoWrite	=	1	X	X	X
WrStatus	=	0	0	0	0
WrX	=	0	0	0	1
WrY	=	0	0	1	0
WrZ	=	0	0	1	1
WrW	=	0	1	0	0
BadInput	=	0	1	0	1
		# 0	1	1	0
		# 0	1	1	1

Explanation of symbols:
1 = logic High
0 = logic Low
X = don't care

Input Fsbc Address State Definitions

FsbcAdrState	=	InFsbcAdr2	InFsbcAdr1	InFsbcAdr0
WrFsbcRegStatus	=	0	0	0
WrFsbcRegX	=	0	0	1
WrFsbcRegY	=	0	1	0
WrFsbcRegZ	=	0	1	1
WrFsbcRegW	=	1	0	0
WrFsbcReg5	=	1	0	1
WrFsbcReg6	=	1	1	0
WrFsbcReg7	=	1	1	1

```
Equations

     NClrLsw := NClrLowWord;

State Diagram FsbcControlState

FsbcIdle:
          case ( InputMode == Reset )         : FsbcIdle;
               ( InputMode == HmsWrBuf )      : FsbcIdle;
               ( InputMode == HmsWrPack )     : FsbcIdle;
               ( InputMode == HmsWrPackBuf )  : FsbcIdle;
               ( InputMode == FsbcIsReady )   : ReadyState;
          endcase;

ReadyState:
          case ( InputMode == Reset )         : FsbcIdle;
               ( InputMode == HmsWrBuf )      : FsbcWrState;
               ( InputMode == HmsWrPack )     : FsbcPackState;
               ( InputMode == HmsWrPackBuf )  : FsbcWrPackState;
               ( InputMode == FsbcIsReady )   : ReadyState;
          endcase;

FsbcWrState: goto ReadyState;

FsbcWrPackState: goto FsbcPackState;

FsbcPackState: goto FsbcIdle;

FsbcUndefState4: goto FsbcIdle;
FsbcUndefState5: goto FsbcIdle;
FsbcUndefState6: goto FsbcIdle;
FsbcUndefState8: goto FsbcIdle;
FsbcUndefStateA: goto FsbcIdle;
FsbcUndefStateB: goto FsbcIdle;
FsbcUndefStateC: goto FsbcIdle;
FsbcUndefStateD: goto FsbcIdle;
FsbcUndefStateE: goto FsbcIdle;
FsbcUndefStateF: goto FsbcIdle;
```

```
State Diagram FsbcAdrState

WrFsbcRegStatus :
          case ( InputRegVec == NoWrite ) : WrFsbcRegStatus;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == BadInput) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegX :
          case ( InputRegVec == NoWrite ) : WrFsbcRegX;
               ( InputRegVec == BadInput) : WrFsbcRegX;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegY :
          case ( InputRegVec == NoWrite ) : WrFsbcRegY;
               ( InputRegVec == BadInput) : WrFsbcRegY;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegZ :
          case ( InputRegVec == NoWrite ) : WrFsbcRegZ;
               ( InputRegVec == BadInput) : WrFsbcRegZ;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
```

```
                ( InputRegVec == WrW      ) : WrFsbcRegW;
             endcase;

WrFsbcRegW :
           case ( InputRegVec == NoWrite ) : WrFsbcRegW;
                ( InputRegVec == BadInput) : WrFsbcRegW;
                ( InputRegVec == WrStatus) : WrFsbcRegStatus;
                ( InputRegVec == WrX      ) : WrFsbcRegX;
                ( InputRegVec == WrY      ) : WrFsbcRegY;
                ( InputRegVec == WrZ      ) : WrFsbcRegZ;
                ( InputRegVec == WrW      ) : WrFsbcRegW;
             endcase;

WrFsbcReg5: goto WrFsbcRegStatus;
WrFsbcReg6: goto WrFsbcRegStatus;
WrFsbcReg7: goto WrFsbcRegStatus;
```

A.3 Pixel Processor Array Loader State Machine

Table A.6: FSBC to Pixel Processor Data Transfer Control

Signal	*Pin #*	*Pin Definitions*
clk	Pin 1	
NReset	Pin 2	Reset
FsbcAB	Pin 3	MBS or SM request
NFsbcPreq	Pin 4	FSBC data ready
PixPreq	Pin 5	Pixel Processor Ready
BcDone	Pin 6	Bus Controller done
HmsIack	Pin 7	Bitslice acknowledge
NSDPPL	Pin 8	Hms Reads Mux
NDyPPdat	Pin 9	Bitslice does a write to PPdat
chip ground	Pin 10	
chip enable	Pin 11	
NHmsIrq	Pin 12	This request is for the Bitslice
NPixPack	Pin 13	Pixel Processor acknowledge
NIncPix	Pin 14	Increment Pix Proc address
NIncFsbc	Pin 15	Increment Fsbc address
NHmsPPwe	Pin 16	Allow the PP to do a write
NPixWrEnable	Pin 17	Enables Pix Proc for writing
Cycle	Pin 18	State Variable
MuxSel	Pin 19	High select LSW from FSBC
chip VCC	Pin 20	

State Definitions for Data Transfer Machine:

Increment States

IncState	=	NIncPix	NIncFsbc
IncBoth	=	0	0
IncPix	=	0	1
IncFsbc	=	1	0
DontInc	=	1	1

Pack States

PackState	=	NPixPack
PackPix	=	0
NoPack	=	1

Cycle States

InCycle	=	0
NoCycle	=	1

PalReset States

DoReset	=	0
DontReset	=	1

HmsIrq States

DoInt	=	0
DontInt	=	1

PixWrEnable States

DoWrPix	=	0
DontWrPix	=	1

LoadCntrlState	=	Cycle	NHmsIrq	NPixWrEnable	IncState	PackState
Idle	=	NoCycle	DontInt	DontWrPix	DontInc	NoPack
PixTranDontInc	=	InCycle	DontInt	DoWrPix	DontInc	NoPack
PixTranIncBoth	=	InCycle	DontInt	DoWrPix	IncBoth	NoPack
PixTranIncPix	=	InCycle	DontInt	DoWrPix	IncPix	NoPack
PixTranIncFsbc	=	InCycle	DontInt	DoWrPix	IncFsbc	NoPack
NOTE: In the states below, the IncPPadr Pin is used as a state bit. This means that the PPADR counter increments while the HMS has control of the pixel processors. The outputs from the PPADR counter, however, are not enabled on the PP address bus.						
HmsTranMsw	=	InCycle	DoInt	DontWrPix	IncPix	NoPack
HmsTranLsw	=	InCycle	DoInt	DontWrPix	DontInc	NoPack
HmsTranIncFsbc	=	InCycle	DoInt	DontWrPix	IncFsbc	NoPack
PackPixelProc	=	InCycle	DontInt	DontWrPix	DontInc	PackPix

Multiplexer Select States

SelMsw	=	0
SelLsw	=	1

Request Modes

RequestMode	=	PixPreq	NFsbcPreq	FsbcAB
PPReq	=	0	0	0
HmsReq	=	X	0	1
ReqDontCare	=	X	X	X

Hms Input Modes

HmsMode	=	NSDPPL	HmsIack
HmsDone	=	X	1
HmsRead	=	0	0
HmsWait	=	1	0
HmsDontCare	=	X	X

Fsbc Mode

BcMode	=	BcDone
BcLastFetch	=	1
BcDontCare	=	X
BcNotLastFetch	=	0

InputMode	=	NReset	RequestMode	HmsMode	BcMode
Reset	=	0	ReqDontCare	HmsDontCare	BcDontCare
ReqPPTr	=	1	PPReq	HmsDontCare	BcDontCare
ReqHmsTr	=	1	HmsReq	HmsDontCare	BcDontCare
HmsTranDone	=	1	ReqDontCare	HmsDone	BcDontCare
HmsTranWait	=	1	ReqDontCare	HmsWait	BcDontCare
HmsTranRead	=	1	ReqDontCare	HmsRead	BcDontCare
BcTranLastFetch	=	1	ReqDontCare	HmsDontCare	BcLastFetch
BcTranNotLastFetch	=	1	ReqDontCare	HmsDontCare	BcNotLastFetch
InputDontCare	=	1	ReqDontCare	HmsDontCare	BcDontCare

```
Equations

     NHmsPPwe := NDyPPdat;

State Diagram LoadCntrlState

Idle:   !MuxSel := ( RequestMode == HmsReq );

        case (InputMode == Reset )     : Idle;
             (InputMode == ReqPPTr)    : PixTranIncFsbc;
             (InputMode == ReqHmsTr )  : HmsTranMsw;
        endcase;

PixTranIncFsbc:

        MuxSel := 0;
        if ( InputMode == Reset ) then Idle
        else PixTranIncPix;

        case (InputMode == Reset )               : Idle;
             (InputMode == BcTranLastFetch )     : PixTranIncPix;
             (InputMode == BcTranNotLastFetch )  : PixTranIncPix;
        endcase;

PixTranIncPix:

        MuxSel := 1;
        case (InputMode == Reset )               : Idle;
             (InputMode == BcTranLastFetch )     : PackPixelProc;
             (InputMode == BcTranNotLastFetch )  : PixTranIncBoth;
        endcase;

PixTranIncBoth:

        MuxSel := 0;
        goto PixTranIncPix;

PackPixelProc : goto Idle;
```

```
HmsTranMsw:

        MuxSel := !NSDPPL;
        case ( InputMode == Reset    )     : Idle;
             ( InputMode == HmsTranDone ) : Idle;
             ( InputMode == HmsTranRead ) : HmsTranLsw;
             ( InputMode == HmsTranWait ) : HmsTranMsw;
        endcase;

HmsTranLsw:

        MuxSel := NSDPPL;
        case ( InputMode == Reset    )     : Idle;
             ( InputMode == HmsTranDone ) : Idle;
             ( InputMode == HmsTranRead ) : HmsTranIncFsbc;
             ( InputMode == HmsTranWait ) : HmsTranLsw;
        endcase;

HmsTranIncFsbc:

        MuxSel := 0;
        goto HmsTranMsw;
```

A.4 Address Generator for the Endpoint/Color FSBCs

Table A.7: Address Generator for Fifo Stack Bus Controllers

Signal	*Pin #*	*Address Generator Inputs*
clk	Pin 1	Clk Pin
NReset	Pin 2	Reset
NIncFsbcAdr	Pin 3	Increment FSBC address
PPLpack	Pin 4	Input only, not used
NPixPack	Pin 5	Input only, not used
NC6	Pin 6	Input only, not used
NC7	Pin 7	Input only, not used
NC8	Pin 8	Input only, not used
NC9	Pin 9	Input only, not used
chip ground	Pin 10	
chip enable	Pin 11	
Address Generator Outputs		
ResetOut	Pin 12	Used for test vectors only
NOutFsbcPack	Pin 13	Packet acknowledge to Fsbc's
FsbcAdr0	Pin 14	Fsbc address bit 0
FsbcAdr1	Pin 15	Fsbc address bit 1
FsbcAdr2	Pin 16	Fsbc address bit 2
FsbcSel	Pin 17	Selects Color or Endpoint Fsbc
BcDone	Pin 18	Last address out
NC19	Pin 19	Input or combinatorial output
chip vcc	Pin 20	

FsbcAddressState	=	FsbcSel	FsbcAdr2	FsbcAdr1	FsbcAdr0
EndRdReg0	=	0	0	0	0
EndRdReg1	=	0	0	0	1
EndRdReg2	=	0	0	1	0
EndRdReg3	=	0	0	1	1
EndRdReg4	=	0	1	0	0
EndRdReg5	=	0	1	0	1
EndRdReg6	=	0	1	1	0
EndRdReg7	=	0	1	1	1
ColRdReg0	=	1	0	0	0
ColRdReg1	=	1	0	0	1
ColRdReg2	=	1	0	1	0
ColRdReg3	=	1	0	1	1
ColRdReg4	=	1	1	0	0
ColRdReg5	=	1	1	0	1
ColRdReg6	=	1	1	1	0
ColRdReg7	=	1	1	1	1

Input Mode Definitions

InputMode	=	ResetOut	NIncFsbcAdr
Reset	=	0	X
IncFsbcAdr	=	1	0
Wait	=	1	1

WhoGivesPack	=	PPLpack	NPixPack
HmsGivesPack	=	1	1
PPALoaderGivesPack	=	0	0

Table A.8: FSBC Address Generator

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

```
Equations

     BcDone := ( FsbcAddressState == ColRdReg3 );

     !ResetOut =  ( PPLpack  )
                # (!NPixPack )
                # (!NReset   );

     !NOutFsbcPack := ( WhoGivesPack == HmsGivesPack )
                    #( WhoGivesPack == PPALoaderGivesPack );

State Diagram FsbcAddressState

EndRdReg0 :
                    case ( InputMode == Reset )      : EndRdReg0;
                         ( InputMode == Wait )       : EndRdReg0;
                         ( InputMode == IncFsbcAdr ) : EndRdReg1;
                    endcase;

EndRdReg1 :
                    case ( InputMode == Reset )      : EndRdReg0;
                         ( InputMode == Wait )       : EndRdReg1;
                         ( InputMode == IncFsbcAdr ) : EndRdReg2;
                    endcase;

EndRdReg2 :
                    case ( InputMode == Reset )      : EndRdReg0;
                         ( InputMode == Wait )       : EndRdReg2;
                         ( InputMode == IncFsbcAdr ) : EndRdReg3;
                    endcase;

EndRdReg3:
                    case ( InputMode == Reset )      : EndRdReg0;
                         ( InputMode == Wait )       : EndRdReg3;
                         ( InputMode == IncFsbcAdr ) : ColRdReg4;
                    endcase;

EndRdReg4: Goto EndRdReg0;
EndRdReg5: Goto EndRdReg0;
```

```
EndRdReg6: Goto EndRdReg0;
EndRdReg7: Goto EndRdReg0;
ColRdReg0: Goto EndRdReg0;

ColRdReg4 :
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg4;
                     ( InputMode == IncFsbcAdr )  : ColRdReg2;
                endcase;

ColRdReg2 :
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg2;
                     ( InputMode == IncFsbcAdr )  : ColRdReg3;
                endcase;

ColRdReg3:
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg3;
                     ( InputMode == IncFsbcAdr )  : EndRdReg0;
                endcase;

ColRdReg1: Goto EndRdReg0;
ColRdReg5: Goto EndRdReg0;
ColRdReg6: Goto EndRdReg0;
ColRdReg7: Goto EndRdReg0;
```

A.5 Address Generator for Pixel Processors

Table A.9: Pixel Processor Address Generator

Signal	*Pin #*	*Pixel Processor Address Generator Definitions*
clk	Pin 1	Clock Pin
NReset	Pin 2	Reset
NIncPixAdr	Pin 3	Increment Pixel Processor address
NPixPack	Pin 4	Indicates loader done
NC5	Pin 5	not used
NC6	Pin 6	not used
NC7	Pin 7	not used
PPLpack	Pin 8	Reset when a PPLpack is given
NC9	Pin 9	input only, Not Used
chip ground	Pin 10	
chip enable	Pin 11	
NC12	Pin 12	not used
NC13	Pin 13	not used
PPAdr0	Pin 14	PP address bit 0
PPAdr1	Pin 15	PP address bit 1
PPAdr2	Pin 16	PP address bit 2
PPAdr3	Pin 17	PP address bit 3
PPAdr4	Pin 18	PP address bit 4
PPAdr5	Pin 19	PP address bit 5
chip vcc	Pin 20	

Data Transfer Machine State Definitions

PPAddr	=	PPAdr3	PPAdr2	PPAdr1	PPAdr0
PP15	=	1	1	1	1
PP14	=	1	1	1	0
PP13	=	1	1	0	1
PP12	=	1	1	0	0
PP11	=	1	0	1	1
PP10	=	1	0	1	0
PP9	=	1	0	0	1
PP8	=	1	0	0	0
PP7	=	0	1	1	1
PP6	=	0	1	1	0
PP5	=	0	1	0	1
PP4	=	0	1	0	0
PP3	=	0	0	1	1
PP2	=	0	0	1	0
PP1	=	0	0	0	1
PP0	=	0	0	0	0

Pixel Processor Addresses

PPAddressState	=	PPAddr
WrReg1	=	PP1
WrReg2	=	PP2
WrReg3	=	PP3
WrReg4	=	PP4
WrReg5	=	PP5
WrReg6	=	PP6
WrReg7	=	PP7
WrReg8	=	PP8
WrReg9	=	PP9
WrReg10	=	PP10
WrReg11	=	PP11
WrReg12	=	PP12
WrReg13	=	PP13
WrReg14	=	PP14
WrReg15	=	PP15
WrReg16	=	PP16

Input Mode Definitions

InputMode	=	NReset	NPixPack	PPLpack	NIncPixAdr
Reset	=	0	X	X	X
		# 1	0	X	X
		# 1	X	1	X
IncPix	=	1	1	0	0
Wait	=	1	1	0	1

NOTE: The counter resets when a *PixPack is given, this happens during transfers from the FSBC to the Pixel Processors. The counter also resets when PPLpack is given. This is needed because during the time that the HMS has control of the Pixel Processors, the bit that increments the ppaddress sometimes asserted. Therefore at the end when control is given back to the PPAloader, the PPADR needs to be reset to 0.

```
Equations

     PPAdr4 := 0;
     PPAdr5 := 0;

State Diagram PPAddressState

WrReg0 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg0;
                    ( InputMode == IncPix ) : WrReg3;
               endcase;

WrReg1 : goto WrReg0;

WrReg2 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg2;
                    ( InputMode == IncPix ) : WrReg5;
               endcase;

WrReg3 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg3;
                    ( InputMode == IncPix ) : WrReg2;
               endcase;

WrReg4 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg4;
                    ( InputMode == IncPix ) : WrReg7;
               endcase;

WrReg5 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg5;
                    ( InputMode == IncPix ) : WrReg4;
               endcase;
```

```
WrReg6 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg6;
                     ( InputMode == IncPix ) : WrReg9;
                endcase;

WrReg7 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg7;
                     ( InputMode == IncPix ) : WrReg6;
                endcase;

WrReg8 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg8;
                     ( InputMode == IncPix ) : WrReg11;
                endcase;

WrReg9 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg9;
                     ( InputMode == IncPix ) : WrReg8;
                endcase;

WrReg10 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg10;
                     ( InputMode == IncPix ) : WrReg13;
                endcase;

WrReg11 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg11;
                     ( InputMode == IncPix ) : WrReg10;
                endcase;

WrReg12 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg12;
```

```
                    ( InputMode == IncPix ) : WrReg0;
               endcase;

WrReg13:
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg13;
                    ( InputMode == IncPix ) : WrReg12;
               endcase;

WrReg14 : goto WrReg0;

WrReg15 : goto WrReg0;
```

A.6 Address Generator for Endpoint/Color FSBC

Table A.10: Address Generator for Fifo Stack Bus Controllers

Signal	*Pin #*	*Address Generator Inputs*
clk	Pin 1	Clk Pin
NReset	Pin 2	Reset
NIncFsbcAdr	Pin 3	Increment FSBC address
PPLpack	Pin 4	Input only, not used
NPixPack	Pin 5	Input only, not used
NC6	Pin 6	Input only, not used
NC7	Pin 7	Input only, not used
NC8	Pin 8	Input only, not used
NC9	Pin 9	Input only, not used
chip ground	Pin 10	
chip enable	Pin 11	
Address Generator Outputs		
ResetOut	Pin 12	Used for test vectors only
NOutFsbcPack	Pin 13	Packet acknowledge to Fsbc's
FsbcAdr0	Pin 14	Fsbc address bit 0
FsbcAdr1	Pin 15	Fsbc address bit 1
FsbcAdr2	Pin 16	Fsbc address bit 2
FsbcSel	Pin 17	Selects Color or Endpoint Fsbc
BcDone	Pin 18	Last address out
NC19	Pin 19	Input or combinatorial output
chip vcc	Pin 20	

FsbcAddressState	=	FsbcSel	FsbcAdr2	FsbcAdr1	FsbcAdr0
EndRdReg0	=	0	0	0	0
EndRdReg1	=	0	0	0	1
EndRdReg2	=	0	0	1	0
EndRdReg3	=	0	0	1	1
EndRdReg4	=	0	1	0	0
EndRdReg5	=	0	1	0	1
EndRdReg6	=	0	1	1	0
EndRdReg7	=	0	1	1	1
ColRdReg0	=	1	0	0	0
ColRdReg1	=	1	0	0	1
ColRdReg2	=	1	0	1	0
ColRdReg3	=	1	0	1	1
ColRdReg4	=	1	1	0	0
ColRdReg5	=	1	1	0	1
ColRdReg6	=	1	1	1	0
ColRdReg7	=	1	1	1	1

Input Mode Definitions

InputMode	=	ResetOut	NIncFsbcAdr
Reset	=	0	X
IncFsbcAdr	=	1	0
Wait	=	1	1

WhoGivesPack	=	PPLpack	NPixPack
HmsGivesPack	=	1	1
PPALoaderGivesPack	=	0	0

Table A.11: Data Transfer Machine State Descriptions

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

```
Equations

     BcDone := ( FsbcAddressState == ColRdReg3 );

     !ResetOut =  ( PPLpack  )
                # (!NPixPack )
                # (!NReset   );

     !NOutFsbcPack :=   ( WhoGivesPack == HmsGivesPack )
     #( WhoGivesPack == PPALoaderGivesPack );

State Diagram FsbcAddressState

EndRdReg0 :
                case ( InputMode == Reset )      : EndRdReg0;
                     ( InputMode == Wait )       : EndRdReg0;
                     ( InputMode == IncFsbcAdr ) : EndRdReg1;
                endcase;

EndRdReg1 :
                case ( InputMode == Reset )      : EndRdReg0;
                     ( InputMode == Wait )       : EndRdReg1;
                     ( InputMode == IncFsbcAdr ) : EndRdReg2;
                endcase;

EndRdReg2 :
                case ( InputMode == Reset )      : EndRdReg0;
                     ( InputMode == Wait )       : EndRdReg2;
                     ( InputMode == IncFsbcAdr ) : EndRdReg3;
                endcase;

EndRdReg3:
                case ( InputMode == Reset )      : EndRdReg0;
                     ( InputMode == Wait )       : EndRdReg3;
                     ( InputMode == IncFsbcAdr ) : ColRdReg4;
                endcase;

EndRdReg4: Goto EndRdReg0;
EndRdReg5: Goto EndRdReg0;
```

```
EndRdReg6: Goto EndRdReg0;
EndRdReg7: Goto EndRdReg0;
ColRdReg0: Goto EndRdReg0;

ColRdReg4 :
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg4;
                     ( InputMode == IncFsbcAdr )  : ColRdReg2;
                endcase;

ColRdReg2 :
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg2;
                     ( InputMode == IncFsbcAdr )  : ColRdReg3;
                endcase;

ColRdReg3:
                case ( InputMode == Reset )       : EndRdReg0;
                     ( InputMode == Wait )        : ColRdReg3;
                     ( InputMode == IncFsbcAdr )  : EndRdReg0;
                endcase;

ColRdReg1: Goto EndRdReg0;
ColRdReg5: Goto EndRdReg0;
ColRdReg6: Goto EndRdReg0;
ColRdReg7: Goto EndRdReg0;
```

A.7 HA PixelProcessor Hit Box Tester

Table A.12: HA PixelProcessor Hit Box Tester

Signal	*Pin #*	*HA PixelProcessor Hit Box Tester Inputs*
Clk	Pin 1	
Nwr	Pin 2	lo-true memory write strobe
Nras	Pin 3	row address strobe
Ncas	Pin 4	column address strobe
XGreaterThan	Pin 5	> output from X - comparator
XLessThan	Pin 6	< output from X - comparator
YGreaterThan	Pin 7	> output from Y - comparator
YLessThan	Pin 8	< output from Y - comparator
Nreset	Pin 9	reset for the sampler
Noe	Pin 11	output enable
HA PixelProcessor Hit Box Tester Outputs		
holdX	Pin 12	
NholdX	Pin 13	enable for PPA X-address (Cas) latch 0 → hold, 1 → transparent
holdY	Pin 18	
NholdY	Pin 19	enable for PPA Y-address (Ras) latch 0 → hold, 1 → transparent
HBsel0	Pin 14	selects one of two hitbox corners 0 → Top Right, 1 → Bottom Left
s0,s1	Pin 15, 16	extra state bit
HitDetect	Pin 17	hit detect output to Bitslice status

Table A.13: HA PixelProcessor Hit Box Tester State Descriptions

HBstate	=	HBsel0	s1	s0
idle	=	1	1	1
CmpTopRight1	=	0	1	0
CmpTopRight2	=	0	1	1
CmpBottomLeft1	=	1	1	0
CmpBottomLeft2	=	1	0	1
donewait	=	1	0	0
hitwait	=	0	0	0
undef1	=	0	0	1
HBmode	=	Nwr	Nras	Ncas
quiet	=	1	X	X
incycle	=	0	0	X
done	=	X	1	1

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

```
Equations

Create the flip-flops for RAS & CAS latch control

     holdY =   ! ( (HBstate == donewait) # NholdY # !Nreset );
     NholdY =  ! ( (!Nras & !Nwr & (HBstate == idle)) # holdY );

     holdX =   ! ( (HBstate == donewait) # NholdX # !Nreset );
     NholdX =  ! ( (!Ncas & !Nwr & holdY) # holdX );

Signal a hit detect when inside the 'hitwait' state

     HitDetect := ( HBstate == hitwait );

State Diagram HBstate

Idle:   if ( HBmode == incycle ) then CmpTopRight1 else idle;

CmpTopRight1:   goto CmpTopRight2;

CmpTopRight2:   if ( XLessThan # YGreaterThan ) then donewait
                else CmpBottomLeft1;

CmpBottomLeft1: goto CmpBottomLeft2;

CmpBottomLeft2: if ( XGreaterThan # YLessThan ) then donewait
                else hitwait;

donewait:       if ( HBmode == done ) then idle else donewait;

hitwait:        if ( !Nreset ) then idle else hitwait;

undef1:         goto idle;
```

A.8 Input FSBC Controller

Table A.14: Input FSBC Controller

Signal	*Pin #*	*Input FSBC Controller Inputs*
clk	Pin 1	
NReset	Pin 2	Reset State Machine
NInFsbcPreq	Pin 3	Fsbc is ready for input
HmsInPack	Pin 4	Hms is done writing Fsbc
NDyInFsbc	Pin 5	Hms writes Fsbc buffer register
FsbcReg0	Pin 6	Fsbc Register address bit 0
FsbcReg1	Pin 7	Fsbc Register address bit 1
FsbcReg2	Pin 8	Fsbc Register address bit 2
NClrLowWord	Pin 9	Clr Fsbc Low word before input
chip ground	Pin 10	
chip enable	Pin 11	
Input FSBC Controller Output Latches		
NFsbcReady	Pin 12	Ready signal for Hms
NFsbcWrite	Pin 13	InFsbc write strobe
NFsbcPack	Pin 14	Packet acknowledge for InFsbc
InFsbcAdr0	Pin 15	Fsbc Reg address
InFsbcAdr1	Pin 16	Fsbc Reg address
InFsbcAdr2	Pin 17	Fsbc Reg address
NClrLsw	Pin 18	Buffer has data for MBS
State0	Pin 19	State bit for controller
chip vcc	Pin 20	

FsbcControlState	=	State0	NFsbcReady	NFsbcWrite	NFsbcPack	
FsbcIdle	=	0	1	1	1	The IDLE state
FsbcWrState	=	0	0	0	1	Write a register in the FSBC
ReadyState	=	0	0	1	1	Let the HMS know that the FSBC is ready
FsbcPackState	=	0	0	1	0	Send PACK to the InputFsbc
FsbcWrPackState	=	1	0	0	1	Write a register in the FSBC and this state always followed by packstate
FsbcUndefState4	=	0	1	0	0	
FsbcUndefState5	=	0	1	0	1	
FsbcUndefState6	=	0	1	1	0	
FsbcUndefState8	=	1	0	0	0	
FsbcUndefStateA	=	1	0	1	0	
FsbcUndefStateB	=	1	0	1	1	
FsbcUndefStateC	=	1	1	0	0	
FsbcUndefStateD	=	1	1	0	1	
FsbcUndefStateE	=	1	1	1	0	
FsbcUndefStateF	=	1	1	1	1	

Table A.15: Input Fsbc Controller State Descriptions

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

Fsbc Controller Input Signal Modes

InputMode	=	NReset	NInFsbcPreq	HmsInPack	NDyInFsbc
Reset	=	0	X	X	X
FsbcIsReady	=	1	0	0	1
HmsWrBuf	=	1	0	0	0
HmsWrPack	=	1	0	1	1
HmsWrPackBuf	=	1	0	1	0
DontCare	=	1	X	X	X

Input Vector for the Fsbc Register Address

InputRegVec	=	NDyInFsbc	FsbcReg2	FsbcReg1	FsbcReg0
NoWrite	=	1	X	X	X
WrStatus	=	0	0	0	0
WrX	=	0	0	0	1
WrY	=	0	0	1	0
WrZ	=	0	0	1	1
WrW	=	0	1	0	0
BadInput	=	0	1	0	1
#	=	0	1	1	0
#	=	0	1	1	1

Explanation of symbols:
1 = logic High
0 = logic Low
X = don't care

Input Fsbc Address State Definitions

FsbcAdrState	=	InFsbcAdr2	InFsbcAdr1	InFsbcAdr0
WrFsbcRegStatus	=	0	0	0
WrFsbcRegX	=	0	0	1
WrFsbcRegY	=	0	1	0
WrFsbcRegZ	=	0	1	1
WrFsbcRegW	=	1	0	0
WrFsbcReg5	=	1	0	1
WrFsbcReg6	=	1	1	0
WrFsbcReg7	=	1	1	1

```
Equations

     NClrLsw := NClrLowWord;

State Diagram FsbcControlState

FsbcIdle:
          case ( InputMode == Reset )         : FsbcIdle;
               ( InputMode == HmsWrBuf )      : FsbcIdle;
               ( InputMode == HmsWrPack )     : FsbcIdle;
               ( InputMode == HmsWrPackBuf )  : FsbcIdle;
               ( InputMode == FsbcIsReady )   : ReadyState;
          endcase;

ReadyState:
          case ( InputMode == Reset )         : FsbcIdle;
               ( InputMode == HmsWrBuf )      : FsbcWrState;
               ( InputMode == HmsWrPack )     : FsbcPackState;
               ( InputMode == HmsWrPackBuf )  : FsbcWrPackState;
               ( InputMode == FsbcIsReady )   : ReadyState;
          endcase;

FsbcWrState: goto ReadyState;

FsbcWrPackState: goto FsbcPackState;

FsbcPackState: goto FsbcIdle;

FsbcUndefState4: goto FsbcIdle;
FsbcUndefState5: goto FsbcIdle;
FsbcUndefState6: goto FsbcIdle;
FsbcUndefState8: goto FsbcIdle;
FsbcUndefStateA: goto FsbcIdle;
FsbcUndefStateB: goto FsbcIdle;
FsbcUndefStateC: goto FsbcIdle;
FsbcUndefStateD: goto FsbcIdle;
FsbcUndefStateE: goto FsbcIdle;
FsbcUndefStateF: goto FsbcIdle;
```

```
State Diagram FsbcAdrState

WrFsbcRegStatus :
          case ( InputRegVec == NoWrite ) : WrFsbcRegStatus;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == BadInput) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegX :
          case ( InputRegVec == NoWrite ) : WrFsbcRegX;
               ( InputRegVec == BadInput) : WrFsbcRegX;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegY :
          case ( InputRegVec == NoWrite ) : WrFsbcRegY;
               ( InputRegVec == BadInput) : WrFsbcRegY;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
               ( InputRegVec == WrW     ) : WrFsbcRegW;
          endcase;

WrFsbcRegZ :
          case ( InputRegVec == NoWrite ) : WrFsbcRegZ;
               ( InputRegVec == BadInput) : WrFsbcRegZ;
               ( InputRegVec == WrStatus) : WrFsbcRegStatus;
               ( InputRegVec == WrX     ) : WrFsbcRegX;
               ( InputRegVec == WrY     ) : WrFsbcRegY;
               ( InputRegVec == WrZ     ) : WrFsbcRegZ;
```

```
                ( InputRegVec == WrW      ) : WrFsbcRegW;
            endcase;

    WrFsbcRegW :
            case ( InputRegVec == NoWrite ) : WrFsbcRegW;
                 ( InputRegVec == BadInput) : WrFsbcRegW;
                 ( InputRegVec == WrStatus) : WrFsbcRegStatus;
                 ( InputRegVec == WrX      ) : WrFsbcRegX;
                 ( InputRegVec == WrY      ) : WrFsbcRegY;
                 ( InputRegVec == WrZ      ) : WrFsbcRegZ;
                 ( InputRegVec == WrW      ) : WrFsbcRegW;
              endcase;

    WrFsbcReg5: goto WrFsbcRegStatus;
    WrFsbcReg6: goto WrFsbcRegStatus;
    WrFsbcReg7: goto WrFsbcRegStatus;
```

A.9 Pixel Processor Address Generator

Table A.16: Pixel Processor Address Generator

Signal	*Pin #*	*Pixel Processor Address Generator Definitions*
clk	Pin 1	Clock Pin
NReset	Pin 2	Reset
NIncPixAdr	Pin 3	Increment Pixel Processor address
NPixPack	Pin 4	Indicates loader done
NC5	Pin 5	not used
NC6	Pin 6	not used
NC7	Pin 7	not used
PPLpack	Pin 8	Reset when a PPLpack is given
NC9	Pin 9	input only, Not Used
chip ground	Pin 10	
chip enable	Pin 11	
DMA Machine Output Latches		
NC12	Pin 12	not used
NC13	Pin 13	not used
PPAdr0	Pin 14	PP address bit 0
PPAdr1	Pin 15	PP address bit 1
PPAdr2	Pin 16	PP address bit 2
PPAdr3	Pin 17	PP address bit 3
PPAdr4	Pin 18	PP address bit 4
PPAdr5	Pin 19	PP address bit 5
chip vcc	Pin 20	

Table A.17: Data Transfer Machine State Definitions

Data Transfer Machine State Definitions

PPAddr	=	PPAdr3	PPAdr2	PPAdr1	PPAdr0
PP15	=	1	1	1	1
PP14	=	1	1	1	0
PP13	=	1	1	0	1
PP12	=	1	1	0	0
PP11	=	1	0	1	1
PP10	=	1	0	1	0
PP9	=	1	0	0	1
PP8	=	1	0	0	0
PP7	=	0	1	1	1
PP6	=	0	1	1	0
PP5	=	0	1	0	1
PP4	=	0	1	0	0
PP3	=	0	0	1	1
PP2	=	0	0	1	0
PP1	=	0	0	0	1
PP0	=	0	0	0	0

Pixel Processor Addresses

PPAddressState	=	PPAddr
WrReg1	=	PP1
WrReg2	=	PP2
WrReg3	=	PP3
WrReg4	=	PP4
WrReg5	=	PP5
WrReg6	=	PP6
WrReg7	=	PP7
WrReg8	=	PP8
WrReg9	=	PP9
WrReg10	=	PP10
WrReg11	=	PP11
WrReg12	=	PP12
WrReg13	=	PP13
WrReg14	=	PP14
WrReg15	=	PP15
WrReg16	=	PP16

Input Mode Definitions

InputMode	=	NReset	NPixPack	PPLpack	NIncPixAdr
Reset	=	0	X	X	X
#	=	1	0	X	X
#	=	1	X	1	X
IncPix	=	1	1	0	0
Wait	=	1	1	0	1

NOTE: The counter resets when a *PixPack is given, this happens during transfers from the FSBC to the Pixel Processors. The counter also resets when PPLpack is given. This is needed because during the time that the HMS has control of the Pixel Processors, the bit that increments the ppaddress is asserted. Therefore at the end when control is given back to the PPAloader, the PPADR needs to be reset to 0.

```
Equations

     PPAdr4 := 0;
     PPAdr5 := 0;

State Diagram PPAddressState

WrReg0 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg0;
                    ( InputMode == IncPix ) : WrReg3;
               endcase;

WrReg1 : goto WrReg0;

WrReg2 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg2;
                    ( InputMode == IncPix ) : WrReg5;
               endcase;

WrReg3 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg3;
                    ( InputMode == IncPix ) : WrReg2;
               endcase;

WrReg4 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg4;
                    ( InputMode == IncPix ) : WrReg7;
               endcase;

WrReg5 :
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg5;
                    ( InputMode == IncPix ) : WrReg4;
               endcase;
```

```
WrReg6 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg6;
                     ( InputMode == IncPix ) : WrReg9;
                endcase;

WrReg7 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg7;
                     ( InputMode == IncPix ) : WrReg6;
                endcase;

WrReg8 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg8;
                     ( InputMode == IncPix ) : WrReg11;
                endcase;

WrReg9 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg9;
                     ( InputMode == IncPix ) : WrReg8;
                endcase;

WrReg10 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg10;
                     ( InputMode == IncPix ) : WrReg13;
                endcase;

WrReg11 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg11;
                     ( InputMode == IncPix ) : WrReg10;
                endcase;

WrReg12 :
                case ( InputMode == Reset  ) : WrReg0;
                     ( InputMode == Wait   ) : WrReg12;
```

```
                    ( InputMode == IncPix ) : WrReg0;
               endcase;

WrReg13:
               case ( InputMode == Reset  ) : WrReg0;
                    ( InputMode == Wait   ) : WrReg13;
                    ( InputMode == IncPix ) : WrReg12;
               endcase;

WrReg14 : goto WrReg0;

WrReg15 : goto WrReg0;
```

A.10 Pixel Processor Array Loader State Machine

Table A.18: FSBC to Pixel Processor Data Transfer Control

Signal	*Pin #*	*Pin Definitions*
clk	Pin 1	
NReset	Pin 2	Reset
FsbcAB	Pin 3	MBS or SM request
NFsbcPreq	Pin 4	FSBC data ready
PixPreq	Pin 5	Pixel Processor Ready
BcDone	Pin 6	Bus Controller done
HmsIack	Pin 7	Bitslice acknowledge
NSDPPL	Pin 8	Hms Reads Mux
NDyPPdat	Pin 9	Bitslice does a write to PPdat
chip ground	Pin 10	
chip enable	Pin 11	
DMA Machine Output Latches		
NHmsIrq	Pin 12	This request is for bitslice
NPixPack	Pin 13	Pixel Processor acknowledge
NIncPix	Pin 14	Increment Pix Proc address
NIncFsbc	Pin 15	Increment Fsbc address
NHmsPPwe	Pin 16	Allow the PP to do a write
NPixWrEnable	Pin 17	Enables Pix Proc for writing
Cycle	Pin 18	State Variable
MuxSel	Pin 19	High select LSW from FSBC
chip VCC	Pin 20	

State Definitions for Data Transfer Machine:

Increment States

IncState	=	NIncPix	NIncFsbc
IncBoth	=	0	0
IncPix	=	0	1
IncFsbc	=	1	0
DontInc	=	1	1

Pack States

PackState	=	NPixPack
PackPix	=	0
NoPack	=	1

Cycle States

InCycle	=	0
NoCycle	=	1

PalReset States

DoReset	=	0
DontReset	=	1

HmsIrq States

DoInt	=	0
DontInt	=	1

PixWrEnable States

DoWrPix	=	0
DontWrPix	=	1

Multiplexer Select States

SelMsw	=	0
SelLsw	=	1

LoadCntrlState	=	Cycle	NHmsIrq	NPixWrEnable	IncState	PackState
Idle	=	NoCycle	DontInt	DontWrPix	DontInc	NoPack
PixTranDontInc	=	InCycle	DontInt	DoWrPix	DontInc	NoPack
PixTranIncBoth	=	InCycle	DontInt	DoWrPix	IncBoth	NoPack
PixTranIncPix	=	InCycle	DontInt	DoWrPix	IncPix	NoPack
PixTranIncFsbc	=	InCycle	DontInt	DoWrPix	IncFsbc	NoPack
NOTE: In the states below, the IncPPadr Pin is used as a state bit. This means that the PPADR counter increments while the HMS has control of the pixel processors. The outputs from the PPADR counter, however, are not enabled on the PP address bus.						
HmsTranMsw	=	InCycle	DoInt	DontWrPix	IncPix	NoPack
HmsTranLsw	=	InCycle	DoInt	DontWrPix	DontInc	NoPack
HmsTranIncFsbc	=	InCycle	DoInt	DontWrPix	IncFsbc	NoPack
PackPixelProc	=	InCycle	DontInt	DontWrPix	DontInc	PackPix

Request Modes

RequestMode	=	PixPreq	NFsbcPreq	FsbcAB
PPReq	=	0	0	0
HmsReq	=	X	0	1
ReqDontCare	=	X	X	X

Hms Input Modes

HmsMode	=	NSDPPL	HmsIack
HmsDone	=	X	1
HmsRead	=	0	0
HmsWait	=	1	0
HmsDontCare	=	X	X

Fsbc Mode

BcMode	=	BcDone
BcLastFetch	=	1
BcDontCare	=	X
BcNotLastFetch	=	0

InputMode	=	NReset	RequestMode	HmsMode	BcMode
Reset	=	0	ReqDontCare	HmsDontCare	BcDontCare
ReqPPTr	=	1	PPReq	HmsDontCare	BcDontCare
ReqHmsTr	=	1	HmsReq	HmsDontCare	BcDontCare
HmsTranDone	=	1	ReqDontCare	HmsDone	BcDontCare
HmsTranWait	=	1	ReqDontCare	HmsWait	BcDontCare
HmsTranRead	=	1	ReqDontCare	HmsRead	BcDontCare
BcTranLastFetch	=	1	ReqDontCare	HmsDontCare	BcLastFetch
BcTranNotLastFetch	=	1	ReqDontCare	HmsDontCare	BcNotLastFetch
InputDontCare	=	1	ReqDontCare	HmsDontCare	BcDontCare

```
Equations

     NHmsPPwe := NDyPPdat;

State Diagram LoadCntrlState

Idle:   !MuxSel := ( RequestMode == HmsReq );

        case (InputMode == Reset )     : Idle;
             (InputMode == ReqPPTr)    : PixTranIncFsbc;
             (InputMode == ReqHmsTr )  : HmsTranMsw;
        endcase;

PixTranIncFsbc:

        MuxSel := 0;
        if ( InputMode == Reset ) then Idle
        else PixTranIncPix;

        case (InputMode == Reset )               : Idle;
             (InputMode == BcTranLastFetch )     : PixTranIncPix;
             (InputMode == BcTranNotLastFetch )  : PixTranIncPix;
        endcase;

PixTranIncPix:

        MuxSel := 1;
        case (InputMode == Reset )               : Idle;
             (InputMode == BcTranLastFetch )     : PackPixelProc;
             (InputMode == BcTranNotLastFetch )  : PixTranIncBoth;
        endcase;

PixTranIncBoth:

        MuxSel := 0;
        goto PixTranIncPix;

PackPixelProc : goto Idle;
```

```
HmsTranMsw:

        MuxSel := !NSDPPL;
        case ( InputMode == Reset    )      : Idle;
             ( InputMode == HmsTranDone ) : Idle;
             ( InputMode == HmsTranRead ) : HmsTranLsw;
             ( InputMode == HmsTranWait ) : HmsTranMsw;
        endcase;

HmsTranLsw:

        MuxSel := NSDPPL;
        case ( InputMode == Reset    )      : Idle;
             ( InputMode == HmsTranDone ) : Idle;
             ( InputMode == HmsTranRead ) : HmsTranIncFsbc;
             ( InputMode == HmsTranWait ) : HmsTranLsw;
        endcase;

HmsTranIncFsbc:

        MuxSel := 0;
        goto HmsTranMsw;
```

A.11 Pixel Processor Handshake State Machine

Table A.19: Fix for the PP Handshake

Signal	*Pin #*	*Pin Definitions*
clk	Pin 1	PAL clock input
NReset	Pin 2	Reset
Ti	Pin 3	TakenIn from Pixel Processor
NFwt	Pin 4	Wait signal
NPixPack	Pin 5	Indicates loader done
NHmsPPPack	Pin 6	Acknowledge from Hms
NSlr	Pin 7	Scanline Read from PP
NClearWait	Pin 8	input only, not used
NPPWe	Pin 9	Write PP in next state
chip ground	Pin 10	
chip enable	Pin 11	
State Machine Output Latches		
NHmsPPWe	Pin 12	Pixel Processor Write enable
NPPWaitSlb	Pin 13	Wait the pixel processor
SlbRequest	Pin 14	Start up the SLB
SlbState0	Pin 15	State bit for SLB
State0	Pin 16	State bit 0
State1	Pin 17	State bit 1
NNPR	Pin 18	*NPR, PP new packet request
NNPA	Pin 19	*NPA, PP new packet acknowledge
chip vcc	Pin 20	

Table A.20: Scanline Buffer Input Mode Pins

SlbInMode	=	NReset	NFwt	NSlr	NClearWait
SlbReset	=	0	X	X	X
SlbNoReq	=	1	X	1	X
SlbReq	=	1	X	0	X
GoSlb	=	1	1	X	X
WaitSlb	=	1	0	X	X
Release	=	1	X	X	0
HoldWait	=	1	X	X	1

Scanline Monitor States

SlbMonitor	=	NPPWaitSlb	SlbState0
MonitorIdle	=	1	0
GotReq	=	1	1
PPWaitForSlb	=	0	0
SlbUndefState1	=	0	1

PP Handshake State Machine Input Mode Definitions

Pack	=	NPixPack	NHmsPPPack
Ack	=	0	X
#	=	X	0
NoAck	=	1	1
AckDontCare	=	X	X

HandShakeInput	=	NReset	Ti	NFwt	Pack	
HandShakeReset	=	0	X	X	X	
GotA	=	1	0	X	Ack	Pack only
GotAT	=	1	1	1	Ack	Pack and Taken In
GotATW	=	1	1	0	Ack	Pack, Taken In and Wait
WaitForA	=	1	X	X	NoAck	Wait for PACK
GotT	=	1	1	1	AckDontCare	Taken In
GotTW	=	1	1	0	AckDontCare	Taken In and Wait
WaitForT	=	1	0	X	AckDontCare	Wait for Taken In
WaitForW	=	1	X	0	AckDontCare	Wait for wait
WaitReleased	=	1	X	1	AckDontCare	Wait negated

Handshake State Machine States

HandShakeState	=	State1	State0	NNPR	NNPA	
PPrequest	=	1	1	0	1	State # 13
HaveA	=	1	1	1	1	State # 15
HaveATW	=	1	0	1	1	State # 11
PPacknowledge	=	1	0	1	0	State # 10
PPWait	=	0	1	1	1	State # 7
UndefState0	=	0	0	0	0	
UndefState1	=	0	0	0	1	
UndefState2	=	0	0	1	0	
UndefState3	=	0	0	1	1	
UndefState4	=	0	1	0	0	
UndefState5	=	0	1	0	1	
UndefState6	=	0	1	1	0	
UndefState8	=	1	0	0	0	
UndefState9	=	1	0	0	1	
UndefState12	=	1	1	0	0	
UndefState14	=	1	1	1	0	

```
Equations

     NHmsPPWe := NPPWe ;

     SlbRequest := ( !NPPWaitSlb & NClearWait );

State Diagram SlbMonitor

MonitorIdle :

     case ( SlbInMode == SlbReset ) : MonitorIdle;
          ( SlbInMode == SlbNoReq ) : MonitorIdle;
          ( SlbInMode == SlbReq   ) : GotReq;
     endcase;

GotReq :

     case ( SlbInMode == SlbReset ) : MonitorIdle;
          ( SlbInMode == GoSlb    ) : PPWaitForSlb;
          ( SlbInMode == WaitSlb  ) : GotReq;
     endcase;

PPWaitForSlb :

     case ( SlbInMode == SlbReset ) : MonitorIdle;
          ( SlbInMode == Release  ) : MonitorIdle;
          ( SlbInMode == HoldWait ) : PPWaitForSlb;
     endcase;

SlbUndefState1 : goto MonitorIdle;

State Diagram HandShakeState

PPrequest :
          case  ( HandShakeInput == HandShakeReset ) : PPrequest;
                ( HandShakeInput == GotA           ) : HaveA;
                ( HandShakeInput == GotAT          ) : PPacknowledge;
                ( HandShakeInput == GotATW         ) : HaveATW;
                ( HandShakeInput == WaitForA       ) : PPrequest;
```

```
            endcase;

HaveA :
            case  ( HandShakeInput == HandShakeReset ) : PPrequest;
                  ( HandShakeInput == GotT           ) : PPacknowledge;
                  ( HandShakeInput == GotTW          ) : HaveATW;
                  ( HandShakeInput == WaitForT       ) : HaveA;
            endcase;

HaveATW :
            case  ( HandShakeInput == HandShakeReset ) : PPrequest;
                  ( HandShakeInput == WaitReleased   ) : PPacknowledge;
                  ( HandShakeInput == WaitForW       ) : HaveATW;
            endcase;

PPacknowledge :
            case  ( HandShakeInput == HandShakeReset ) : PPrequest;
                  ( HandShakeInput == WaitForW       ) : PPWait;
                  ( HandShakeInput == WaitReleased   ) : PPrequest;
            endcase;


PPWait :
            case  ( HandShakeInput == HandShakeReset ) : PPrequest;
                  ( HandShakeInput == WaitForW       ) : PPWait;
                  ( HandShakeInput == WaitReleased   ) : PPrequest;
            endcase;

UndefState0  : goto PPrequest;
UndefState1  : goto PPrequest;
UndefState2  : goto PPrequest;
UndefState3  : goto PPrequest;
UndefState4  : goto PPrequest;
UndefState5  : goto PPrequest;
UndefState6  : goto PPrequest;
UndefState8  : goto PPrequest;
UndefState9  : goto PPrequest;
UndefState12 : goto PPrequest;
UndefState14 : goto PPrequest;
```

A.12 P.P. Address Generator for the Scanline Buffer State Machine

Table A.21: PP Address Generator for the ScanLine Buffer

Signal	*Pin #*	*Address Generator Inputs*
clk	Pin 1	PAL clock input
NReset	Pin 2	Reset Pin
NIncPPadr	Pin 3	Increments counter
NC4	Pin 4	input only, not used
NC5	Pin 5	input only, not used
NC6	Pin 6	input only, not used
NC7	Pin 7	input only, not used
NC8	Pin 8	input only, not used
NC9	Pin 9	input only, not used
chip ground	Pin 10	
chip enable	Pin 11	
Address Generator Outputs		
NLast16Loaded	Pin 12	Indicate Msw loaded
PPadr0	Pin 13	PP address bit 0
PPadr1	Pin 14	PP address bit 1
PPadr2	Pin 15	PP address bit 2
PPadr3	Pin 16	PP address bit 3
PPadr4	Pin 17	PP address bit 4
PPadr5	Pin 18	PP address bit 5
NFirst16Loaded	Pin 19	Indicate Lsw loaded
chip vcc	Pin 20	

Input Mode Bits for Address Generator

SlbAdrMode	=	NReset	NIncPPadr
Reset	=	0	X
Wait	=	1	1
Count	=	1	0

Pixel Processor Address Pins

ColorSelect	=	PPadr3	PPadr2
Blu	=	0	0
Grn	=	0	1
Red	=	1	0
Win	=	1	1

BitSelect	=	PPadr1	PPadr0
selbits01	=	0	0
selbits23	=	0	1
selbits45	=	1	0
selbits67	=	1	1

PPAddress	=	PPadr3	PPadr2	PPadr1	PPadr0

SlbAdrState	=	ColorSelect	BitSelect
SlbAdrBlu01	=	Blu	selbits01
SlbAdrBlu23	=	Blu	selbits23
SlbAdrBlu45	=	Blu	selbits45
SlbAdrBlu67	=	Blu	selbits67
SlbAdrGrn01	=	Grn	selbits01
SlbAdrGrn23	=	Grn	selbits23
SlbAdrGrn45	=	Grn	selbits45
SlbAdrGrn67	=	Grn	selbits67
SlbAdrRed01	=	Red	selbits01
SlbAdrRed23	=	Red	selbits23
SlbAdrRed45	=	Red	selbits45
SlbAdrRed67	=	Red	selbits67
SlbAdrWin01	=	Win	selbits01
SlbAdrWin23	=	Win	selbits23
SlbAdrWin45	=	Win	selbits45
SlbAdrWin67	=	Win	selbits67

```
Equations

     !NFirst16Loaded = ( PPAddress ==  & 0111);
     !NLast16Loaded  = ( PPAddress ==  & 1111);

State Diagram SlbAdrState

SlbAdrBlu01:
             case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                  ( SlbAdrMode == Wait  ) : SlbAdrBlu01;
                  ( SlbAdrMode == Count ) : SlbAdrBlu23;
             endcase;

SlbAdrBlu23:
             case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                  ( SlbAdrMode == Wait  ) : SlbAdrBlu23;
                  ( SlbAdrMode == Count ) : SlbAdrBlu45;
```

```
                endcase;

    SlbAdrBlu45:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrBlu45;
                     ( SlbAdrMode == Count ) : SlbAdrBlu67;
                endcase;

    SlbAdrBlu67:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrBlu67;
                     ( SlbAdrMode == Count ) : SlbAdrGrn01;
                endcase;

    SlbAdrGrn01:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrGrn01;
                     ( SlbAdrMode == Count ) : SlbAdrGrn23;
                endcase;

    SlbAdrGrn23:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrGrn23;
                     ( SlbAdrMode == Count ) : SlbAdrGrn45;
                endcase;

    SlbAdrGrn45:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrGrn45;
                     ( SlbAdrMode == Count ) : SlbAdrGrn67;
                endcase;

    SlbAdrGrn67:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrGrn67;
                     ( SlbAdrMode == Count ) : SlbAdrRed01;
                endcase;

    SlbAdrRed01:
```

```
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrRed01;
                     ( SlbAdrMode == Count ) : SlbAdrRed23;
                endcase;

    SlbAdrRed23:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrRed23;
                     ( SlbAdrMode == Count ) : SlbAdrRed45;
                endcase;

    SlbAdrRed45:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrRed45;
                     ( SlbAdrMode == Count ) : SlbAdrRed67;
                endcase;

    SlbAdrRed67:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrRed67;
                     ( SlbAdrMode == Count ) : SlbAdrWin01;
                endcase;

    SlbAdrWin01:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrWin01;
                     ( SlbAdrMode == Count ) : SlbAdrWin23;
                endcase;

    SlbAdrWin23:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrWin23;
                     ( SlbAdrMode == Count ) : SlbAdrWin45;
                endcase;

    SlbAdrWin45:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrWin45;
                     ( SlbAdrMode == Count ) : SlbAdrWin67;
```

```
                endcase;

    SlbAdrWin67:
                case ( SlbAdrMode == Reset ) : SlbAdrBlu01;
                     ( SlbAdrMode == Wait  ) : SlbAdrWin67;
                     ( SlbAdrMode == Count ) : SlbAdrBlu01;
                endcase;
```

A.13 Scanline Buffer Controller State Machine

Table A.22: Scan Line Buffer Controller Pin Assignments

Signal	*Pin #*	*Address Generator Inputs*
clk	Pin 1	PAL clock input
NReset	Pin 2	Reset Pin
NSlbReq	Pin 3	Scanline buffer request
NHmsRead	Pin 4	Hms reads scanline buffer
NFirst16Loaded	Pin 5	Indicates to transfer to pipe reg
NLast16Loaded	Pin 6	Indicates to alert HMS
NC7	Pin 7	Input only, not used
NPPpack	Pin 8	Hms is sending PPpack
BcDone	Pin 9	Bus controller is done
chip ground	Pin 10	
chip enable	Pin 11	
Address Generator Outputs		
NHmsPPpack	Pin 12	Buffered NPPpack
NClearWait	Pin 13	Clears PP wait FF
NIncPPadr	Pin 14	Increment address in Address PAL
NSlbReady	Pin 15	Indicate data is ready in buffer
NPclkEnable	Pin 16	Transfer from Shadow to Pipe Reg clk
NDclkEnable	Pin 17	Shift in clk enable
NMode	Pin 18	AM29818 mode input
BcDoneB	Pin 19	Buffered BcDone
chip vcc	Pin 20	

Address PAL Count Bit

DoInc	=	0
DoNotInc	=	1

Scan Line Buffer Ready Bit

SlbIsReady	=	0
SlbNotReady	=	1

AM29818 Control

Control818	=	NMode	NDclkEnable	NPclkEnable
Idle818	=	1	1	1
ShiftIn	=	0	0	1
ShadowToPipe	=	1	1	0
Nop818	=	0	1	1

**PPWaitSlb FF Clear Bit*

ClearPPWait	=	0
DontClearPPWait	=	1

Scan Line Buffer Input Mode

SlbMode	=	NReset	NSlbReq	NHmsRead	NFirst16Loaded	NLast16Loaded
Reset	=	0	X	X	X	X
NoReq	=	1	0	X	X	X
SlbReq	=	1	1	1	1	1
HmsReadSlb	=	1	1	0	X	X
LswLoaded	=	1	1	1	0	1
MswLoaded	=	1	1	1	1	0

SlbState	=	NSlbReady	Control818	NIncPPadr
SlbIdle	=	SlbNotReady	Idle818	DoNotInc
SlbShiftIn	=	SlbNotReady	ShiftIn	DoInc
SlbTranToPipe0	=	SlbNotReady	ShadowToPipe	DoNotInc
SlbTranToPipe1	=	SlbIsReady	ShadowToPipe	DoNotInc
SlbWaitLsw	=	SlbIsReady	Nop818	DoNotInc
SlbWaitMsw	=	SlbIsReady	Idle818	DoNotInc

SlbIdle	=	1	1	1	1	1
SlbShiftIn	=	1	1	0	1	0
SlbTranToPipe0	=	1	0	1	0	1
SlbTranToPipe1	=	0	0	1	0	1
SlbWaitLsw	=	0	0	1	1	1
SlbWaitMsw	=	0	1	1	1	1

```
Equations

   NHmsPPpack := NPPpack;

   BcDoneB    := BcDone;

State Diagram SlbState

SlbIdle : NClearWait := 1;
           case ( SlbMode == Reset     ) : SlbIdle;
                ( SlbMode == SlbReq    ) : SlbShiftIn;
                ( SlbMode == HmsReadSlb ) : SlbIdle;
                ( SlbMode == LswLoaded  ) : SlbIdle;
                ( SlbMode == MswLoaded  ) : SlbIdle;
          endcase;

SlbShiftIn : NClearWait := 1;
          case ( SlbMode == Reset     ) : SlbIdle;
               ( SlbMode == SlbReq    ) : SlbShiftIn;
               ( SlbMode == HmsReadSlb ) : SlbIdle;
               ( SlbMode == LswLoaded  ) : SlbTranToPipe0;
               ( SlbMode == MswLoaded  ) : SlbWaitLsw;
          endcase;
```

```
SlbTranToPipe0 :      NClearWait := 1;

          if ( SlbMode == Reset ) then SlbIdle else SlbShiftIn;

SlbWaitLsw :  NClearWait := 1;
          case ( SlbMode == Reset       ) : SlbIdle;
               ( SlbMode == SlbReq      ) : SlbWaitLsw;
               ( SlbMode == HmsReadSlb  ) : SlbTranToPipe1;
               ( SlbMode == LswLoaded   ) : SlbWaitLsw;
               ( SlbMode == MswLoaded   ) : SlbWaitLsw;
          endcase;

SlbTranToPipe1 :      NClearWait := 1;

          if ( SlbMode == Reset ) then SlbIdle else SlbWaitMsw;

          case ( SlbMode == Reset       ) : SlbIdle;
               ( SlbMode == SlbReq      ) : SlbTranToPipe1;
               ( SlbMode == HmsReadSlb  ) : SlbWaitMsw;
               ( SlbMode == LswLoaded   ) : SlbTranToPipe1;
               ( SlbMode == MswLoaded   ) : SlbTranToPipe1;
          endcase;

SlbWaitMsw :  NClearWait := NHmsRead;
          case ( SlbMode == Reset       ) : SlbIdle;
               ( SlbMode == SlbReq      ) : SlbWaitMsw;
               ( SlbMode == HmsReadSlb  ) : SlbIdle;
               ( SlbMode == LswLoaded   ) : SlbWaitMsw;
               ( SlbMode == MswLoaded   ) : SlbWaitMsw;
          endcase;

SlbClearPPWait : goto SlbIdle;
```

A.14 *Sync Generator for Shadowfax VLSI

Table A.23: System *SYNC Signal Generation

Signal	*Pin #*	*Address Generator Inputs*
clk	Pin 1	PAL clock input
NReset	Pin 2	PAL reset Pin
NC3	Pin 3	input only, not used
NC4	Pin 4	input only, not used
NC5	Pin 5	input only, not used
NC6	Pin 6	input only, not used
NC7	Pin 7	input only, not used
NC8	Pin 8	input only, not used
NC9	Pin 9	input only, not used
NC10	Pin 10	input only, not used
chip ground	Pin 12	
NC13	Pin 13	input only, not used
Address Generator Outputs		
NC14	Pin 14	output latch, or input, not used
NC15	Pin 15	output latch, or input, not used
NC16	Pin 16	output latch, or input, not used
Sync	Pin 17	The Shadowfax Sync signal
cnt0	Pin 18	counter bit 0
cnt1	Pin 19	counter bit 1
cnt2	Pin 20	counter bit 2
cnt3	Pin 21	counter bit 3
cnt4	Pin 22	counter bit 4
NC23	Pin 23	output latch, not used
chip vcc	Pin 24	

Table A.24: PAL Output State Definitions

SyncState	=	Sync	Counter				
			cnt4	cnt3	cnt2	cnt1	cnt0
one	=	0	0	0	0	0	1
two	=	0	0	0	0	1	0
three	=	0	0	0	0	1	1
four	=	0	0	0	1	0	0
five	=	0	0	0	1	0	1
six	=	0	0	0	1	1	0
seven	=	0	0	0	1	1	1
eight	=	0	0	1	0	0	0
nine	=	0	0	1	0	0	1
ten	=	0	0	1	0	1	0
eleven	=	0	0	1	0	1	1
twelve	=	0	0	1	1	0	0
thirteen	=	0	0	1	1	0	1
fourteen	=	0	0	1	1	1	0
fifteen	=	0	0	1	1	1	1
sixteen	=	0	1	0	0	0	0
seventeen	=	0	1	0	0	0	1
eighteen	=	1	1	0	0	1	0
nineteen	=	0	1	0	0	1	1

Table A.25: PAL Output State Definitions Continued

SyncState	=	Sync	Counter				
			cnt4	cnt3	cnt2	cnt1	cnt0
undefstate0	=	0	0	0	0	0	0
undefstate1	=	0	1	0	1	0	0
undefstate2	=	0	1	0	1	0	1
undefstate3	=	0	1	0	1	1	0
undefstate4	=	0	1	0	1	1	1
undefstate5	=	0	1	1	0	0	0
undefstate6	=	0	1	1	0	0	1
undefstate7	=	0	1	1	0	1	0
undefstate8	=	0	1	1	0	1	1
undefstate9	=	0	1	1	1	0	0
undefstate10	=	0	1	1	1	0	1
undefstate11	=	0	1	1	1	1	0
undefstate12	=	0	1	1	1	1	1
undefstate0s	=	1	0	0	0	0	0
undefstate1s	=	1	1	0	1	0	0
undefstate2s	=	1	1	0	1	0	1
undefstate3s	=	1	1	0	1	1	0
undefstate4s	=	1	1	0	1	1	1
undefstate5s	=	1	1	1	0	0	0
undefstate6s	=	1	1	1	0	0	1
undefstate7s	=	1	1	1	0	1	0
undefstate8s	=	1	1	1	0	1	1
undefstate9s	=	1	1	1	1	0	0
undefstate10s	=	1	1	1	1	0	1
undefstate11s	=	1	1	1	1	1	0
undefstate12s	=	1	1	1	1	1	1

State Diagram SyncState

```
one          : if ( NReset == 0 ) then one else two;
two          : if ( NReset == 0 ) then one else three;
three        : if ( NReset == 0 ) then one else four;
four         : if ( NReset == 0 ) then one else five;
five         : if ( NReset == 0 ) then one else six;
six          : if ( NReset == 0 ) then one else seven;
seven        : if ( NReset == 0 ) then one else eight;
eight        : if ( NReset == 0 ) then one else nine;
nine         : if ( NReset == 0 ) then one else ten;
ten          : if ( NReset == 0 ) then one else eleven;
eleven       : if ( NReset == 0 ) then one else twelve;
twelve       : if ( NReset == 0 ) then one else thirteen;
thirteen     : if ( NReset == 0 ) then one else fourteen;
fourteen     : if ( NReset == 0 ) then one else fifteen;
fifteen      : if ( NReset == 0 ) then one else sixteen;
sixteen      : if ( NReset == 0 ) then one else seventeen;
seventeen    : if ( NReset == 0 ) then one else eighteen;
eighteen     : if ( NReset == 0 ) then one else one;

nineteen     : goto one;

undefstate0  : goto one;
undefstate1  : goto one;
undefstate2  : goto one;
undefstate3  : goto one;
undefstate4  : goto one;
undefstate5  : goto one;
undefstate6  : goto one;
undefstate7  : goto one;
undefstate8  : goto one;
undefstate9  : goto one;
undefstate10 : goto one;
undefstate11 : goto one;
undefstate12 : goto one;

undefstate0s: goto one;
undefstate1s: goto one;
```

```
undefstate2s: goto one;
undefstate3s: goto one;
undefstate4s: goto one;
undefstate5s: goto one;
undefstate6s: goto one;
undefstate7s: goto one;
undefstate8s: goto one;
undefstate9s: goto one;
undefstate10s: goto one;
undefstate11s: goto one;
undefstate12s: goto one;
```

A.15 HA Pixel Processor Hit Box Tester

Table A.26: HA PixelProcessor Hit Box Tester

Signal	*Pin #*	*HA PixelProcessor Hit Box Tester Inputs*
Clk	Pin 1	
Nwr	Pin 2	lo-true memory write strobe
Nras	Pin 3	row address strobe
Ncas	Pin 4	column address strobe
XGreaterThan	Pin 5	> output from X - comparator
XLessThan	Pin 6	< output from X - comparator
YGreaterThan	Pin 7	> output from Y - comparator
YLessThan	Pin 8	< output from Y - comparator
Nreset	Pin 9	reset for the sampler
Noe	Pin 11	output enable
HA PixelProcessor Hit Box Tester Outputs		
holdX	Pin 12	
NholdX	Pin 13	enable for PPA X-address (Cas) latch 0 → hold, 1 → transparent
holdY	Pin 18	
NholdY	Pin 19	enable for PPA Y-address (Ras) latch 0 → hold, 1 → transparent
HBsel0	Pin 14	selects one of two hitbox corners 0 → Top Right, 1 → Bottom Left
s0,s1	Pin 15, 16	extra state bit
HitDetect	Pin 17	hit detect output to Bitslice status

Table A.27: HA PixelProcessor Hit Box Tester State Descriptions

HBstate	=	HBsel0	s1	s0
idle	=	1	1	1
CmpTopRight1	=	0	1	0
CmpTopRight2	=	0	1	1
CmpBottomLeft1	=	1	1	0
CmpBottomLeft2	=	1	0	1
donewait	=	1	0	0
hitwait	=	0	0	0
undef1	=	0	0	1
HBmode	=	Nwr	Nras	Ncas
quiet	=	1	X	X
incycle	=	0	0	X
done	=	X	1	1

Explanation of symbols: 1 = logic High
0 = logic Low
X = don't care

```
Equations

Create the flip-flops for RAS & CAS latch control

     holdY =    ! ( (HBstate == donewait) # NholdY # !Nreset );
     NholdY =   ! ( (!Nras & !Nwr & (HBstate == idle)) # holdY );

     holdX =    ! ( (HBstate == donewait) # NholdX # !Nreset );
     NholdX =   ! ( (!Ncas & !Nwr & holdY) # holdX );

Signal a hit detect when inside the 'hitwait' state

     HitDetect := ( HBstate == hitwait );

State Diagram HBstate

Idle:    if ( HBmode == incycle ) then CmpTopRight1 else idle;

CmpTopRight1:    goto CmpTopRight2;

CmpTopRight2:    if ( XLessThan # YGreaterThan ) then donewait
                 else CmpBottomLeft1;

CmpBottomLeft1: goto CmpBottomLeft2;

CmpBottomLeft2: if ( XGreaterThan # YLessThan ) then donewait
                 else hitwait;

donewait:        if ( HBmode == done ) then idle else donewait;

hitwait:         if ( !Nreset ) then idle else hitwait;

undef1:          goto idle;
```

A.16 HA Processor WIMMED Controller

Table A.28: HA Processor WIMMED Controller Pins

Signal	*Pin #*	*Signal Descriptions*
Clk	Pin 1	
NHoldWim	Pin 5	Requests Immediate Field Register to be sourced in Next State
HAInt	Pin 6	Interrupt from Bitslice
NWRT	Pin 7	Common Bus read/write mode control
CBIIN[3]	Pin 8	Level 3 Interrupt
*CBLDS	Pin 9	Common Bus Lower Data Strobe
*CBADS	Pin 12	Common Bus Address Strobe
Output Pins		
RIMMED	Pin 16	Source Immediate Field Register to Immediate Bus
NRIMMED	Pin 17	"
NVecEn, CBDTA	Pin 19	Enable Interrupt Vector on Common Bus, Generate DTACK

Table A.29: HA Processor WIMMED Controller Input Modes

Vecmode	=	*HAInt*	*NWRT*	*CBI*	*LDS*	*ADS*
Envec	=	1	1	1	0	0
Novec1	=	0	X	X	X	X
Novec2	=	X	0	X	X	X
Novec3	=	X	X	0	X	X
Novec4	=	X	X	X	1	X
Novec5	=	X	X	X	X	1

```
Equations
     NVecEn    = !(HAInt & NWRT & CBI & !LDS & !ADS);
     NRIMMED:  = NHoldWim;
     RIMMED:   = !NHoldWim;
Enable NVecEn = 1;
```

A.17 HA Processor System Clocks

Table A.30: HA Processor System Clocks Pins

Signal	*Pin #*	*Signal Descriptions*
clk	Pin 1	
HaltReq	Pin 2	Request to stop clocks
SSReq	Pin 3	Request to single step
ClkSync	Pin 4	Clock PAL is syncronized to this input
Pclkreq	Pin 5	Request to increment P.C.
	Output Pins	
M10_RawClk	Pin 19	Non-stopable 100ns clock
IENCLK750CYC	Pin 18	ALU destination register write enable
HMSCLK100NS4	Pin 17	Stopable 100ns clock
NClrReq	Pin 16	Auto clear to Input Requests
NStrobe	Pin 15	Disables destination decoders until outputs stablize
pclk100ns	Pin 14	100ns clock to Microsequencer
HMSCLK50NS	Pin 13	Non-stopable 50ns clock
Hms50ns	Pin 12	Stopable 50ns clock

Table A.31: HA Processor System Clocks State Assignments

clkstate	=	*M10_RawClk*	*M20_Clk*	*M10_SysClk*	*NStrobe*	*pclk100ns*
phase0	=	1	1	1	1	1
phase1	=	1	0	1	0	1
phase2	=	0	1	0	0	0
phase3	=	0	0	0	0	0
hltphase1	=	1	0	1	1	1
hltphase2	=	0	1	1	1	1
hltphase3	=	0	0	1	1	1
shftphase2	=	0	1	1	0	0
shftphase3	=	0	0	1	0	0

Table A.32: HA Processor System Clocks Input Modes

clkmode	=	*SSReq*	*HaltReq*	*ShiftReq*
noop	=	X	0	X
SStep	=	1	1	0
Halt	=	0	1	0
Shift	=	0	1	1
Illegal	=	1	1	1

```
Equations

  ND75_IEN := !( ( clkstate == phase2 ) # (clkstate == phase3) );

  NClrReq  := !( ( clkstate == phase0 ) & ( clkmode == SStep )
              # ( clkstate == phase0 ) & ( clkmode == Shift ) );

State_diagram clkstate

phase0:  Hms50nsclk:=   ( clkmode == Shift )
                      # ( clkmode == Halt  );
            case ( clkmode == noop)    : phase1 ;
                 ( clkmode == SStep)   : phase1 ;
                 ( clkmode == Halt )   : hltphase1 ;
                 ( clkmode == Shift)   : phase1 ;
                 ( clkmode == Illegal): hltphase1 ;
            endcase;

phase1:  Hms50nsclk:= 1;
            case ( clkmode == noop )   : phase2 ;
                 ( clkmode == SStep)   : phase2 ;
                 ( clkmode == Halt )   : phase2 ;
                 ( clkmode == Shift)   : shftphase2 ;
                 ( clkmode == Illegal): phase2 ;
            endcase;

phase2:  Hms50nsclk:= 0;
            if ClkSync then phase2 else phase3;

phase3:  Hms50nsclk:= 1;
            if !ClkSync then phase3 else phase0;

hltphase1:  Hms50nsclk:= 1;
            goto hltphase2;

hltphase2:  Hms50nsclk:= 1;
            if ClkSync then hltphase2 else hltphase3;

hltphase3:  Hms50nsclk:= 1;
```

```
            if !ClkSync then hltphase3 else phase0;

shftphase2: Hms50nsclk := 1;
            goto shftphase3;

shftphase3: Hms50nsclk := 1;
            goto phase0 ;
```

A.18 HA Processor DMA Controller

Table A.33: HA Processor DMA Controller Pin Descriptions

Input Signals	*Pin #*	*Signal Descriptions*
clk	Pin 1	
Nreadreq	Pin 2	Request to read Mass Memory
Nwrtreq	Pin 3	Request to write Mass Memory
Ndtackraw	Pin 4	Unsyncronized version of DTACK
cbtimout	Pin 5	Time out period to gain control of the Common Bus
Nigotit	Pin 6	When asserted, Bitslice controls the Common Bus
Nreset	Pin 9	Reset DMA PAL
Output Pin Descriptions		
CB_CBA	Pin 19	Latch data from common bus
NSDtack	Pin 18	Synchronized version of CBDTACK
common	Pin 17	Request the common bus
CBbusy	Pin 16	DMA state machine busy
R_Wmode	Pin 15	Keeps read/write mode
NTimerEn	Pin 14	Start timer bit
NSIgotit	Pin 13	Synchronized version of IGOTITI
CB_GAB	Pin 12	Enable data to common bus

Table A.34: HA Processor DMA Controller Input Modes

CBstate	=	*CBbusy*	*Common*
idle	=	0	0
transfer	=	1	1
DMAdone	=	1	0
undef	=	0	1

Table A.35: HA Processor DMA Controller State Assignments

RQMode	=	*Nwrtreq*	*Nreadreq*	*Nreset*	*cbtimout*	*NSIgotit*	*NSDtack*
reset	=	X	X	0	X	X	X
write	=	0	1	1	X	X	X
read	=	1	0	1	X	X	X
Tout	=	X	X	1	1	0	X
NoBus	=	X	X	1	0	1	X
Nodtack	=	X	X	1	0	0	1
Gotdtack	=	X	X	1	0	0	0
noop	=	1	1	1	X	X	X

```
Equations

   NSIgotit := Nigotit;
   NSDtack  := Ndtackraw;

State_diagram CBstate

idle: R_Wmode  := !Nreadreq;
      NTimerEn := 1;
      CB_CBA   := 0;
      CB_GAB   := 0;

      case ( RQMode == write ) : transfer;
           ( RQMode == read  ) : transfer;
           ( RQMode == noop  ) : idle;
           ( RQMode == reset ) : idle;
      endcase;

transfer: R_Wmode  := R_Wmode;
          NTimerEn := NSIgotit;
          CB_CBA   := ( (RQMode == Gotdtack) & R_Wmode );
          CB_GAB   := ( !NSIgotit & !R_Wmode );

          case ( RQMode == reset )  : idle;
               ( RQMode == Tout  )  : DMAdone;
               ( RQMode == NoBus )  : transfer;
               ( RQMode == Nodtack ): transfer;
               ( RQMode == Gotdtack): DMAdone;
          endcase;

DMAdone:  R_Wmode :=  R_Wmode;
          NTimerEn:=  1;
          CB_CBA  :=  0;
          CB_GAB  :=  0;
              goto idle;

undef: goto idle;
```

A.19 HA Processor Maintenance Register Address Decoder

Table A.36: HA Processor Maintenance Register Address Decoder Pins

Signal	*Pin #*	*Signal Descriptions*
NCBADS	Pin 1	Indicates Address lines are stable
NCBDEV	Pin 2	Upper Address bits = 7FF if asserted
NCBWRT	Pin 3	Read/Write Control
CBADR12	Pin 4	Lower Common Bus Address Lines
CBADR11	Pin 5	
CBADR10	Pin 6	
CBADR9	Pin 7	
CBADR8	Pin 8	
CBADR7	Pin 9	
CBADR6	Pin 11	
CBADR5	Pin 12	
CBADR4	Pin 13	
Output Pins		
NRDMAINT	Pin 16	Enable Maint. Reg to Common Bus
NWRMAINT	Pin 15	Write Maint. Reg from Common Bus

Equations

```
NRDMAINT = !(!NCBADS & !NCBDEV & NCBWRT  & (ADRmode == Maintadr) );

NWRMAINT = !(!NCBADS & !NCBDEV & !NCBWRT & (ADRmode == Maintadr) );
```

Table A.37: HA Processor Maintenance Register Address Decoder State Assignments

ADRmode	=	CBADR12	CBADR11	CBADR10	CBADR9	CBADR8	CBADR7	CBADR6	CBADR5	CBADR4
Maintadr	=	1	0	0	0	0	0	0	1	1
NoMaint1	=	0	X	X	X	X	X	X	X	X
NoMaint2	=	X	1	X	X	X	X	X	X	X
NoMaint3	=	X	X	1	X	X	X	X	X	X
NoMaint4	=	X	X	X	1	X	X	X	X	X
NoMaint5	=	X	X	X	X	1	X	X	X	X
NoMaint6	=	X	X	X	X	X	1	X	X	X
NoMaint7	=	X	X	X	X	X	X	1	X	X
NoMaint8	=	X	X	X	X	X	X	X	0	X
NoMaint9	=	X	X	X	X	X	X	X	X	0

A.20 Frame Buffer RAS and CAS Control

Table A.38: Frame Buffer RAS and CAS Control

CYCLE TYPE	RASBANK ABW	*RBA	RESULT	
IDLE	XXX	X	No RAS or CAS	
REFRESH	XXX	X	RAS all banks,	No CAS
PP READ (or Longread)	XXX	0	RAS banks AWV,	CAS requesting array(s)
	XXX	1	RAS banks BWV,	CAS requesting array(s)
PP WRITE (or LONGWRITE)	000	X	RAS bank V,	CAS requesting array(s)
	101	X	RAS banks AWV,	CAS requesting array(s)
	011	X	RAS banks BWV,	CAS requesting array(s)
	111	X	RAS banks ABWV,	CAS requesting array(s)
VIDEO READ XFER	XXX	X	RAS all banks,	CAS all arrays
FAST CLEAR READ XFER	000	X	RAS bank V,	CAS requesting array(s)
	101	X	RAS banks AWV,	CAS requesting array(s)
	011	X	RAS banks BWV,	CAS requesting array(s)
	111	X	RAS banks ABWV,	CAS requesting array(s)
FAST CLEAR WRITE XFER	000	X	RAS bank V,	CAS requesting array(s)
	101	X	RAS banks AWV,	CAS requesting array(s)
	011	X	RAS banks BWV,	CAS requesting array(s)
	111	X	RAS banks ABWV,	CAS requesting array(s)

A.20.1 State Tables For The Cycle Sequencer Prom

The following tables define the cycle sequencer PROM in PS 390 systems that use the MB81461.

Table A.39: Cycle Sequencer PROM Summary

Input	Cyc #	Cycle	#	Prom States	
0100	4	PPLONGREAD	4	16 → 18 → 09 → 24	
0000	0	PPLONGWRITE	4	1E → 1A → 0B → 00	
1101	D	REFRESH	3	20 → 0C → 01	
0110	6	PPREAD	3	17 → 19 → 23	Note#[1]
				19 → 23	Note#[2]
0010	2	PPWRITE	3	1f → 1B → 25	Note#[1]
				1b → 25	Note#[2]
0111	7	PPREADBREAK	3	17 → 19 → 23	
0011	3	PPWRITEBREAK	3	1f → 1b → 25	
1111	F	IDLE	1	0D	
1110	E	RESET	1	02	
0101	5	FCRDXFER	6	14 → 10 → 03 →	
				21 → 11 → 04	
1011	B	VREADXFER	3	15 → 12 → 05	
0001	1	FCWRXFER	3	22 → 13 → 06	
		Total	31		

Note#[1] If NOT preceded by state 23 or 25
Note#[2] If preceded by state 23 or 25

Table A.40: Current Cycle PPLONGREAD

Current Cycle: PPLONGREAD (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start cycle	*ppfb cycdn
X	16	18	1	1
X	18	09	1	0
X	09	24	1	1
PPLONGREAD	24	16	0	1
PPLONGWRITE	24	1E	0	1
REFRESH	24	20	0	1
PPREAD	24	17	0	1
PPWRITE	24	1F	0	1
[PPREADBREAK]	24	17	0	1 = These cycles cannot
[PPWRITEBREAK]	24	1F	0	1 = follow PPLONGREAD
IDLE	24	0D	0	1
RESET	24	02	0	1
FCRDXFER	24	14	0	1
VREADXFER	24	15	0	1
FCWRXFER	24	22	0	1

Table A.41: Current Cycle PPLONGWRITE

Current Cycle: PPLONGWRITE (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start cycle	*ppfb cycdn
X	1E	1A	1	1
X	1A	0B	1	0
X	0B	00	1	1
PPLONGREAD	00	16	0	1
PPLONGWRITE	00	1E	0	1
REFRESH	00	20	0	1
PPREAD	00	17	0	1
PPWRITE	00	1F	0	1
[PPREADBREAK]	00	17	0	1 = These cycles cannot
[PPWRITEBREAK]	00	1F	0	1 = follow PPLONGWRITE
IDLE	00	0D	0	1
RESET	00	02	0	1
FCRDXFER	00	14	0	1
VREADXFER	00	15	0	1
FCWRXFER	00	22	0	1

Table A.42: Current Cycle REFRESH

Current Cycle: REFRESH (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start cycle	*ppfb cycdn
X	20	0C	1	1
X	0C	01	1	1
PPLONGREAD	01	16	0	1
PPLONGWRITE	01	1E	0	1
REFRESH	01	20	0	1
PPREAD	01	17	0	1
PPWRITE	01	1F	0	1
[PPREADBREAK]	01	17	0	1 = These cycles cannot
[PPWRITEBREAK]	01	1F	0	1 = follow REFRESH
IDLE	01	0D	0	1
RESET	01	02	0	1
FCRDXFER	01	14	0	1
VREADXFER	01	15	0	1
FCWRXFER	01	22	0	1

Table A.43: Current Cycle PPREAD

Current Cycle: PPREAD[a] (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next *Start cycle	State) *ppfb cycdn
X	17	19	1	0
X	19	23	1	1
PPLONGREAD	23	16	0	1
PPLONGWRITE	23	1E	0	1
REFRESH	23	20	0	1
PPREAD	23	19	0	0
PPWRITE	23	1B	0	0
PPREADBREAK	23	17	0	1
PPWRITEBREAK	23	1F	0	1
IDLE	23	0D	0	1
RESET	23	02	0	1
FCRDXFER	23	14	0	1
VREADXFER	23	15	0	1
FCWRXFER	23	22	0	1

[a]If the previous cycle is PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK (causing the previous state to be 23 or 25), the subsequent PPREAD cycle begins at state 19. This condition is the consequence of being in Page Mode where RAS remains down. If the previous cycle is any cycle other than PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK, the subsequent PPREAD cycle begins at state 17.

Table A.44: Current Cycle PPWRITE

Current Cycle: PPWRITE[a] (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next *Start cycle	State) *ppfb cycdn
X	1F	1B	1	0
X	1B	25	1	1
PPLONGREAD	25	16	0	1
PPLONGWRITE	25	1E	0	1
REFRESH	25	20	0	1
PPREAD	25	19	0	0
PPWRITE	25	1B	0	0
PPREADBREAK	25	17	0	1
PPWRITEBREAK	25	1F	0	1
IDLE	25	0d	0	1
RESET	25	02	0	1
FCRDXFER	25	14	0	1
VREADXFER	25	15	0	1
FCWRXFER	25	22	0	1

[a]If the previous cycle is PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK (causing the previous state to be 23 or 25), the subsequent PPWRITE cycle begins at state 1B. This condition is the consequence of being in Page Mode where RAS remains down. If the previous cycle is any cycle other than PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK, the subsequent PPWRITE cycle begins at state 1F.

Table A.45: Current Cycle PPREADBREAK

Current Cycle: PPREADBREAK [a] (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next *Start cycle	State) *ppfb cycdn
X	17	19	1	0
X	19	23	1	1
PPLONGREAD	23	16	0	1
PPLONGWRITE	23	1E	0	1
REFRESH	23	20	0	1
PPREAD	23	19	0	0
PPWRITE	23	1B	0	0
PPREADBREAK	23	17	0	1
PPWRITEBREAK	23	1F	0	1
IDLE	23	0D	0	1
RESET	23	02	0	1
FCRDXFER	23	14	0	1
VREADXFER	23	15	0	1
FCWRXFER	23	22	0	1

[a] The PPREADBREAK and PPWRITEBREAK cycles can occur only where the previous cycle was a PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK cycle during which *ENDPPCYC1A was asserted (indicating the end of Page Mode operation and resulting in RAS being brought up).

Table A.46: Current Cycle PPWRITEBREAK

Current Cycle: PPWRITEBREAK[a] (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next *Start cycle	State) *ppfb cycdn
X	1F	1B	1	0
X	1B	25	1	1
PPLONGREAD	25	16	0	1
PPLONGWRITE	25	1E	0	1
REFRESH	25	20	0	1
PPREAD	25	19	0	0
PPWRITE	25	1B	0	0
PPREADBREAK	25	17	0	1
PPWRITEBREAK	25	1F	0	1
IDLE	25	0D	0	1
RESET	25	02	0	1
FCRDXFER	25	14	0	1
VREADXFER	25	15	0	1
FCWRXFER	25	22	0	1

[a]The PPREADBREAK and PPWRITEBREAK cycles can occur only where the previous cycle was a PPREAD, PPWRITE, PPREADBREAK or PPWRITEBREAK cycle during which *ENDPPCYC1A was asserted (indicating the end of Page Mode operation and resulting in RAS being brought up).

Table A.47: Current Cycle IDLE

Current Cycle: IDLE (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start *ppfb cycle	cycdn
PPLONGREAD	0D	16	0	1
PPLONGWRITE	0D	1E	0	1
REFRESH	0D	20	0	1
PPREAD	0D	17	0	1
PPWRITE	0D	1F	0	1
[PPREADBREAK]	0D	17	0	1 = These cycles cannot
[PPWRITEBREAK]	0D	1F	0	1 = follow IDLE
IDLE	0D	0D	0	1
RESET	0D	02	0	1
FCRDXFER	0D	14	0	1
VREADXFER	0D	15	0	1
FCWRXFER	0D	22	0	1

Table A.48: Current Cycle RESET

Current Cycle: RESET (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start *ppfb cycle	cycdn
PPLONGREAD	02	16	0	1
PPLONGWRITE	02	1E	0	1
REFRESH	02	20	0	1
PPREAD	02	17	0	1
PPWRITE	02	1F	0	1
[PPREADBREAK]	02	17	0	1 = These cycles cannot
[PPWRITEBREAK]	02	1F	0	1 = follow RESET
IDLE	02	0D	0	1
RESET	02	02	0	1
FCRDXFER	02	14	0	1
VREADXFER	02	15	0	1
FCWRXFER	02	22	0	1

Table A.49: Current Cycle FCRDXFER

Current Cycle: FCRDXFER (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start cycle	*ppfb cycdn
X	14	10	1	1
X	10	03	1	1
X	03	21	1	1
X	21	11	1	0
X	11	04	1	1
PPLONGREAD	04	16	0	1
PPLONGWRITE	04	1E	0	1
REFRESH	04	20	0	1
PPREAD	04	17	0	1
PPWRITE	04	1F	0	1
[PPREADBREAK]	04	17	0	1 = These cycles cannot
[PPWRITEBREAK]	04	1F	0	1 = follow FCRDXFER
IDLE	04	0D	0	1
RESET	04	02	0	1
FCRDXFER	04	14	0	1
VREADXFER	04	15	0	1
FCWRXFER	04	22	0	1

Table A.50: Current Cycle VREADXFER

Current Cycle: VREADXFER (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next State) *Start cycle	*ppfb cycdn
X	15	12	1	1
X	12	05	1	1
PPLONGREAD	05	16	0	1
PPLONGWRITE	05	1E	0	1
REFRESH	05	20	0	1
PPREAD	05	17	0	1
PPWRITE	05	1F	0	1
[PPREADBREAK]	05	17	0	1 = These cycles cannot
[PPWRITEBREAK]	05	1F	0	1 = follow VREADXFER
IDLE	05	0D	0	1
RESET	05	02	0	1
FCRDXFER	05	14	0	1
VREADXFER	05	15	0	1
FCWRXFER	05	22	0	1

Table A.51: Current Cycle FCWRXFER

Current Cycle: FCWRXFER (All Values in Hexadecimal)				
Next Cycle	Current Cycle	Next State	(Next *Start cycle	State) *ppfb cycdn
X	22	13	1	0
X	13	06	1	1
PPLONGREAD	06	16	0	1
PPLONGWRITE	06	1E	0	1
REFRESH	06	20	0	1
PPREAD	06	17	0	1
PPWRITE	06	1F	0	1
[PPREADBREAK]	06	17	0	1 = These cycles cannot
[PPWRITEBREAK]	06	1F	0	1 = follow FCWRXFER
IDLE	06	0D	0	1
RESET	06	02	0	1
FCRDXFER	06	14	0	1
VREADXFER	06	15	0	1
FCWRXFER	06	22	0	1