HP 150 Personal Computers

HP 150
TECHNICAL
Reference Manual

Product No. 45625A

HEWLETT PACKARD
Personal Office Computer Division
974 East Arques Avenue, Sunnyvale, CA 94086

Part No. 45625-90001

Printed in U.S.A. 5/84

NOTICE

Purpose: This manual provides detailed technical information on the internals of the HP 150 hardware, firmware, and software. It is designed to aid in the development of hardware and software products which will become part of, or work in conjunction with, the HP 150.

User Level: Use of this information assumes a background in digital logic and assembly-language programming.

Related Products: The *HP 150 Programmer's Tool Kit* (Product 45435A) contains the Assembler, Linker, and appropriate documentation. The *HP 150 MS-DOS User's Guide* (Product 45624A) provides details on MS-DOS.

Compatibility: A hardware/software product designed to be dependent on one specific version of the HP 150 hardware, firmware, or software may be incompatible with other versions of the HP 150 -- or such a product may be incompatible with future personal computer products from Hewlett-Packard, including models based on the HP 150 itself.

Support: Because of the specialized nature of this information, the many capabilities of the HP 150 when used at this level, and the importance of design strategy for successful implementation of hardware/firmware-based products, assistance in the use of this manual is provided through local consulting -- available by the hour (Product 45686A) or by the day (Product 45687A). Contact your local HP Sales and Service Office -- ask for Personal Computer Systems Engineering.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition, and lists the dates of all pages of that edition and all updates. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition.....May 1984

Effective Pages	Date
ALL	May 1984

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

First Edition.....May 1984

PREFACE

CAUTION

This document provides detailed technical information revealing internal details of the HP 150 hardware, firmware, and software.

The information presented allows development of hardware and software products for the HP 150 which may be incompatible with future personal computer products from Hewlett-Packard, including models based upon the HP 150 itself.

This manual presents technical information concerning the HP 150 Personal Computer and covers its hardware, firmware, and software subsystems.

The intent of the manual is to aid in the development of hardware and software products which will become part of, or work in conjunction with, the HP 150.

The reader is directed towards the HP 150 Programmer's Reference materials for complementary programming information including language and development tools.

MANUAL OVERVIEW

This manual consists of the following sections and appendices:

Section 1 - Introduction provides an overview of the HP 150.

Section 2 - Hardware Overview provides product specifications information and briefly describes each subsystem of the HP 150.

Section 3 - Hardware Subsystems contain detailed information on the HP 150's hardware subsystems and helpful design hints.

Section 4 - Memory and I/O Mapping provides information on the memory and input/output bit mappings of the system's processor.

Section 5 - System Software provides information on the operating systems, device drivers, MS-DOS calls, AGIOS, BIOS, configuration, and disc format structure.

Section 6 - System Firmware provides mapping informaiton on system RAM memory.

Section 7 - Programming the HP 150 contains programming information on escape sequences, MS-DOS, AGIOS, alphanumeric and graphics displays, datacomm, keyboard, HPIB, and accessory card interfacing.

Section 8 - AGIOS Function Call Reference provides a reference list of AGIOS function calls.

Appendix A - Logic Diagrams contain schematic diagrams of the HP 150.

Appendix B - Further Reference Documents provides a list of reference documents to supplement this manual.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

INTRODUCTION

SECTION 1

This section provides an overview of the HP 150's system architecture, display, keyboard, data communications, and peripherals.

CONTENTS

HP 150 SYSTEM OVERVIEW

The HP 150 Personal Office Computer, the third member of the Series 100 family, offers a complete business solution and gives the customer more power and memory space in a small, efficient package. The entire sytem (display, processor, keyboard, flexible and Winchester disc drives, and integral printer) occupies only 2.1 square feet of space, about the same "footprint" as an open looseleaf notebook. Several features have been added to decrease the amount of time required to learn and use the system such as a touchscreen interface and an advanced "shell" called the Personal Applictions Manager (P.A.M.) which shields the user from the "computerese" associated with many computer systems.

The HP 150 system features are summarized below.

System Architecture

- Intel 8088 microprocessor (operating at 8 MHz)
- MS-DOS 2.0 operating system
- Built-in HPTouch
- 256K bytes of main memory standard; 640K bytes maximum
- Battery back-up for system configuration and real time clock

The HP 150 uses an Intel 8088 microprocessor running at 8 MHz. The standard system contains 256K bytes of RAM memory for the operating system, applications and user workspace, and can be expanded to 640K. Touchscreen and graphics are standard with the system.

In addition to the above capabilities, the HP 150 is also a customer expandable system. There are two expansion slots, accessible through the back of the unit, which allow the customer to add accessories (such as additional memory) to the system without requiring the assistance of an HP representative or dealer. Installation of accessories is a simple operation, requiring an average of 5 to 10 minutes.

The HP 150 Personal Computer uses MS-DOS 2.0 from Microsoft Corporation as the standard operating system. MS-DOS 2.0 is a single-user, single-task operating system for which many third-party software packages have been developed. The operating system resides on disc and upon initialization MS-DOS is loaded into the processor's main memory.

A unique enhancement by Hewlett-Packard to the MS-DOS operating system has been the addition of an easy-to-understand facility to help the user execute commands. PAM (Personal Applications Manager) provides simple intuitive menus for the most frequently used system commands. Unlike other systems which require the user to learn the system "computerese", PAM through menus and HPTouch, guides the user through commands. With PAM starting applications, creating directories, deleting files and listing existing files can be as easy as touching the screen. For more advanced users the standard MS-DOS command facility is also available.

Display

- Built-in high-resolution on-screen graphics display (512 x 390)
- High-resolution character display; 9 x 14 dot character cells; upper and lower case
- Display enhancements: inverse video, underline, blinking, half-bright, security and all combinations
- Up to two pages of 24 lines x 80 characters of display memory

The HP 150 can display both alphanumeric and graphics on the 9 inch diagonal screen. The alphanumeric display consists of a 27 line by 80 column format. The 25th and 26th lines are used for the screen labeling of function keys (and all are automatically "touchable" through touchscreen), and the 27th line is for system status and error messages. The screen memory stores 2 pages of text, which allows off-screen storage of the display. High resolution characters with true descenders are generated in a 9 x 14 dot cell with half-dot shift. The standard display is green character against a black background.

The graphics display has a resolution of 512 dots horizontally by 390 dots vertically. This gives a 1:1 aspect ratio guaranteeing symmetry (that is, circles look like circles). The numeric keypad also serves as the graphics keypad, allowing the customer to turn on and off the alpha display, turn on and off the graphics display or transfer the graphics display to one of the HP graphics printers. It also displays the graphics cursor and allows it to be moved around the screen.

Keyboard

- Detachable, typewriter-style
- Special editing keys
- Numeric/Graphics pad
- Eight screen-labeled function keys

The HP 150 keyboard is designed to provide a familiar interface to the system and minimize training time. The low-profile keyboard shape, the sculptured keycaps and the dished "home" keys help to make the keyboard comfortable to use. The 107-key keyboard contains the full local editing keys such as cursor control keys, display scrolling keys, "next" and "prev" keys for scrolling by pages and "insert" and "delete" keys for inserting or deleting characters or entire lines.

Series 100 function keys are screen labels used by the system and by application programs to increase the ease of use of the system. With the HP 150, this capability is enhanced by the use of HPTouch. Now, all function keys can be selected by pressing the key itself or by touching the key label on the screen.

Communications/Peripherals

- One RS-232/RS-422 communication port
- One RS-232 communication port
- One HP-IB port
- Full block mode graphics terminal support

Two RS-232-C ports (one of which is capable of RS-422 communication) may be used to connect the system to a remote computer or to serial devices (such as printers or plotters). Flexible protocols allow the use of either hardware or software handshaking and communication speeds can range from 110 to 19,200 baud.

The HP 150 contains the HP 2623 Graphics terminal feature set and can run any HP 3000 software which currently runs on that terminal. This includes block mode for V/3000 software and graphics applications such as HPEasychart and HPDraw as well as line-drawing and math character sets, "security" fields, transmit-only fields, edit checks and Tektronix 4010/4014 emulation.

HARDWARE OVERVIEW

SECTION 2

This section provides product specifications information and briefly describes each module of the HP 150.

CONTENTS

INTRODUCTION

The HP 150 digital hardware performs the logic functions of a 16 bit personal computer with screen graphics. It consists of a Processor PCA, a Memory PCA, a Video subsystem PCA, a Touchscreen PCA, a Front Plane PCA, and a Keyboard PCA within the Keyboard itself. Its operation is based on the 8 MHz 8088 microprocessor.

The Processor PCA provides control signals, input/output and data processing functions. The ROM/RAM PCA provides 256K bytes of dynamic RAM for system and user memory and up to 160K bytes of ROM. The Video subsystem PCA controls the display RAM and provides video display data and timing signals for driving the sweep circuitry. The Touchscreen PCA provides an easy user interface to the system beyond the standard keyboard. Two accessory slots are provided allowing for memory expansion and additional processing and I/O capability.

PRODUCT SPECIFICATIONS

General Description

System Processor:	Intel 8088 microprocessor operating at 8 MHz
Main Memory:	256K bytes of RAM memory
Screen Size:	9 inch diagonal
Alphanumeric	116 X 150 mm (4.5 X 5.9 inches)
Graphics	120 X 160 mm (4.7 X 6.3 inches)
Screen Capacity:	24 lines X 80 columns, 25th and 26th lines for labeling of function keys, 27th line for system status/error messages.
Character Generation:	7 X 10 enhanced dot matrix with 1/2 dot shifting, 9 X 14 dot character cell, noninterlaced raster scan.
Character Size:	1.3 X 2.8 mm (0.05 X 0.11 inches)
Character Set:	Roman8, line drawing, math standard (also bold and italic usable by applications only)
Cursor:	Blinking underline or blinking square

Display Enhancements:	Inverse video, underline, blinking, half-bright, security, and all combinations.
Refresh Rate:	60 Hz
Tube Phosphor:	P31 (green)
Implosion Protection:	Tension band
Keyboard:	Full ASCII code keyboard, eight screen-labeled function keys, auto-repeat, N-key rollover, cursor controls, 18 key numeric pad, detachable with 2.43 m (8 ft.) coiled cable.

Physical Specifications

System Processor Weight:	10.15 kg (22.34 lbs.)
Keyboard Weight:	2.14 kg (4.71lbs.)
Display Monitor Dimensions:	305 mm (W) X 305 mm (D) X 287 mm (H) [12.0 in. X 12.0 in. X 11.3 in.]
Keyboard Dimensions:	
Flat	456 mm (W) X 225 mm (D) X 35 mm (H) [18.0 in. X 8.9 in. X 1.4 in.]
Standing	456 mm (W) X 225 mm (D) X 35 mm (H) [18.0 in. X 8.9 in. X 2.5 in.]

Environmental Conditions

Temperature (Free Space Ambient):	
Non-operating	-40 to +75 C (-40 to +167 F)
Operating	0 to +50 C (+32 to +131 F)
With 2674A (Integral Thermal Printer)	0 to +50 C (+32 to +131 F)
Humidity:	5 to 95% noncondensing
*Vibration:	5-55 Hz @ 0.015" displacement
*Shock	30g, 11 ms, 1/2 sine

*Type tested to qualify for normal shipping and handling original shipping carton.

Product Regulations

This product when used with HP approved options and peripherals meets the requirements of the following agencies/standards for EDP equipment or office equipment in the following countries:

Safety:	Canada - CSA Certification International - IEC 380/435 Compliance United States - U.L. Listing Finland - FEI (pending)
RFI:	Germany - VDE Class B United States - FCC Level B
Datacomm:	CCITT V.24 interchange V.28 electrical Australia - Telecom (pending) Belgium - PTT (pending) Finland - PTT (pending) Germany - FTZ (pending) Sweden - PTT (pending) U. K. - BT (pending)

Power Requirements

Input Voltage:	115 V (+10%,-25%) at 50/60 Hz (+-5%) 230 V (+10%,-25%) at 50/60 Hz (+-5%)
Power Consumption:	
45610A	240 Volt Amp
45650A	356 Volt Amp
45655A	356 Volt Amp
45660A	356 Volt Amp

Communications

Data Channels:	1 HP-IB, 1 RS-232/RS-422, 1 RS-232
HP-IB Channel:	Bus used only for specified HP peripherals
RS-232 Channel:	General asynchronous communications
Data Rate (RS-232):	110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19200 baud

Port 1 and 2:	EIA standard RS-232-C and CCITT V.24; hardware and XON/XOFF handshaking available.
Port 1, only:	RS-422 communication capability

Subsystem Power Requirements

The figures below are the worst case power consumption figures for the main system boards. The values given for accessory slots are obtained by subtracting the main system board power consumption figures from the power available from the power supply printed specifications. A more realistic set of figures for accessory hardware board designers to use in determining power available for accessories is given in Section 3.

PCA	+5 Volt	+12 Volt	-12 Volt
Sweep Board:	125 mA	1.9 A	60 mA
Processor:	2.3 A	140 mA	80 mA
ROM/RAM:	1.0 A	0	0
Video subsystem:	3.4 A	0	0
Front plane:	75 mA	0	0
Thermal Printer:	800 mA	2.2 A *	0
Touch Screen:	0	150 mA	20 mA
Mezzanine Datacomm:	200 mA	50 mA	50 mA
Accessory Slots:	1.75 A	280 mA	270 mA
TOTAL	9.6 A	2.5 A **	480 mA

* This current at +12v for the TPM comes from a separate winding on the power supply.

** This total does not include the current supplied for the TPM on the separate winding.

HP 150 PRINTED CIRCUIT ASSEMBLIES

The HP 150, in its standard configuration consists of seven modules. They are the Processor, Video, Sweep/CRT, Touchscreen, Keyboard, Power Supply, and Front Plane. Below is a block diagram of the HP 150 system showing each of the modules and their associated PCAs.

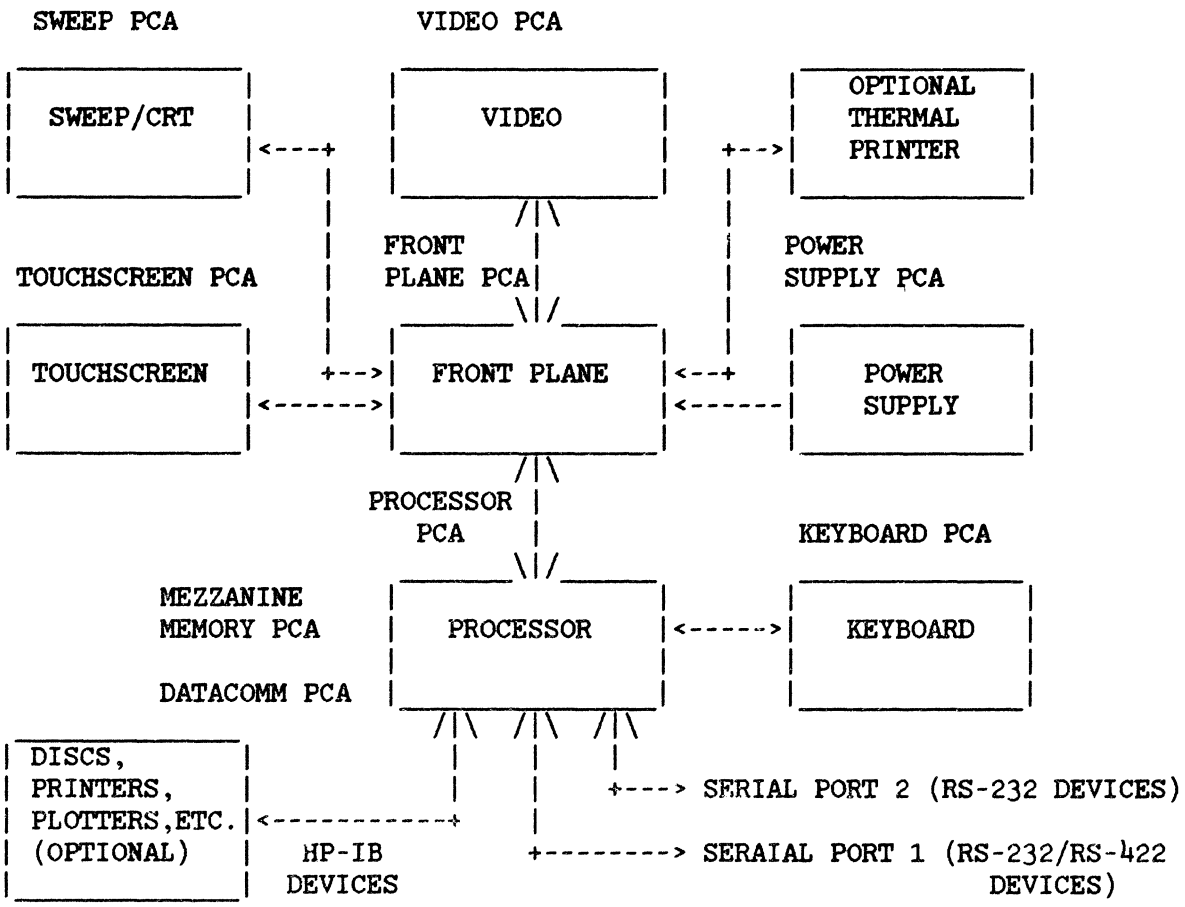


Figure 2-1. HP 150 Block Diagram

A brief description of each PCA in the HP 150 system is given below. Each board is described in detail in Section 3.

Analog Boards

POWER SUPPLY PCA. The power supply used in the HP 150 system is a 120 watt switching supply that provides +5, +12, and -12 volts to the system components. The supply is mounted vertically to the side of the metal chassis and supplies power to the system through a cable which connects to the front plane board and a separate connector on the power supply connects to a ribbon cable to provide +12V to the optional thermal printer (TPM). An overview of the supply is provided in Section 3.

SWEEP PCA. The sweep electronics (assembly 45600-60004) used for HP 150 is the same sweep used in the HP 120 computer system. This board also mounts vertically to the side of the metal chassis and interfaces to the digital logic on the video board via a ribbon cable to the front plane board. This sweep provides focus, brightness, vertical and horizontal centering controls at the rear of the unit. Coarse brightness, horizontal width and vertical size controls are accessible after removal of the shroud of the system. Details about this board as well as the yoke and CRT are found in Section 3.

Digital Logic Boards

There are six PCAs containing primarily digital logic which create the HP 150 hardware personality.

PROCESSOR PCA. The processor board (45611-60002) houses the 8088 microprocessor, the heart of the system. The bulk of the I/O components such as HP-IB controller, keyboard and touchscreen controller, real-time clock, and datacomm are on this board. The processor board interfaces to the rest of the system boards through 96 pin, 60 pin, and 30 pin connectors. The processor board connects to the front plane through the 96 pin connector. The 60 and 30 pin connectors are physically mounted on top of the processor board, and connect the mezzanine memory and datacomm PCAs, respectively. See Section 3 for a discussion on the processor board.

KEYBOARD PCA. The keyboard is detached from the HP 150 unit and a cable connects it to a 6 pin phone jack on the processor board. See Section 3 for more keyboard information.

MEZZANINE MEMORY PCA. The mezzanine memory board (45611-60006) is one of two boards mounted on the processor board in a mezzanine position beneath the video board. The memory board contains ROM, dynamic RAM, CMOS RAM, and indicator LEDs. A 60 pin connector is used to interface the memory and processor boards. See Section 3 for detailed memory board information.

RS232C/422 DATACOMM PCA. The RS232C/422 board (45611-60015) is the other board in a mezzanine position. It connects to the processor board through a 30 pin connector. The board has provision for both asynchronous and synchronous communications and can meet RS232C or RS422 communications standards. Details on this board are in Section 3.

FRONT PLANE PCA. The front plane board (45611-60005) provides the interconnect for the processor, video, touchscreen, sweep, TPM, and accessory boards. The front plane also serves as a conduit for power to the various PCAs from the power supply. Section 3 describes the front plane in more detail.

VIDEO PCA. The alphanumeric and graphics displays are generated by the video subsystem, the core of which is the video board (45611-60003). The video board design is based on an SMC9007 display controller and a custom 40 pin gate array for generation of the alpha and graphics displays respectively. This board is covered in Section 3.

TOUCHSCREEN PCA. The touchscreen board (45611-60001) is actually a mixture of analog and digital circuitry. The 8088 interfaces to the touchscreen via the 8041A keyboard/touchscreen controller. A 10 conductor ribbon cable connects the touchscreen to the front plane where information is in turn exchanged with the processor board. Details on the touchscreen are in Section 3.

HARDWARE SUBSYSTEMS

SECTION 3

This section provides detailed information on the hardware subsystems of the HP 150. Design hints on the optional accessory card are discussed here also. Subsystem discussion includes the front plane, processor, video, keyboard, touchscreen, datacomm, and memory.

CONTENTS

FRONT PLANE

The front plane, or mother board, of the HP 150 system is the board that provides the interconnection of the various components of the system. The processor, video, and two optional accessory boards plug into connectors on the front plane. Four cable connectors interface the power supply, sweep, touchscreen, and optional TPM to the rest of the system via the front plane.

The front plane is used primarily to provide a communications path from the processor board to the video and accessory boards. There is some digital logic on the front plane which is used for transferring bytes from the processor to the TPM. This circuitry is described below. Power for each of the boards comes from the power suppply via the front plane. Video signals are transmitted to the sweep via the front plane and timing signals for the touchscreen are transmitted or received via the front plane.

Thermal (Integral) Printer Interface

The optional thermal printer mechanism (TPM) interfaces to the rest of the system via logic on the front plane and processor boards. The processor sends a data byte to the TPM by writing the byte into a transparent latch on the front plane. A negative going pulse on NTPMWRT is sent causing the latch to receive the data byte. On the positive edge of NTPMWRT (the deassertion of NTPMWRT) the byte is latched into the latch. Also, a D flip-flop on the front plane is clocked which results in NSTR (JF7-15) being asserted (goes low). The assertion of NSTR tells the TPM that a byte is available for it to read. After the TPM reads the byte in the latch, it asserts NACK (JF7-6) by bringing it low. This results in the flip-flop preset inputs going low presetting the two D flip-flops. This makes NSTR go back high in preparation for the next byte to be sent to the TPM by the processor. It also makes the output of the D flip-flop go low which in turn makes NOCINT go low. NOCINT is one of the open collector interrupt lines on the front plane. Asserting NOCINT causes an interrupt to the processor. The interrupt service routine will poll the devices which assert NOCINT to determine which device to service. After receiving an interrupt from the TPM interface logic, the 8088 can send another byte to the TPM. (Note that an interrupt service routine must poll the TPM interface logic last because the poll consists of reading the TPM status which also clears the interrupt at the end of the poll.)

The TPM status can be determined by reading the status register. The NTPMRD signal is asserted (goes low) by the processor board which enables the bus driver on the front plane to drive the data bus. Bit 0=1 if acknowledge has been sent by the TPM in response to a byte sent to the TPM. Bit 1=1 if the TPM is installed and bit 2=1 if paper is present in the TPM. (Bits 3-7 are not used.) The deassertion of NTPMRD (its rising edge) clocks the flip-flop which clears the interrupt generated on NOCINT.

A read or write to the TPM will cause the processor board signal NTPMSL to go low which results in 6 wait states to be added to the bus cycle.

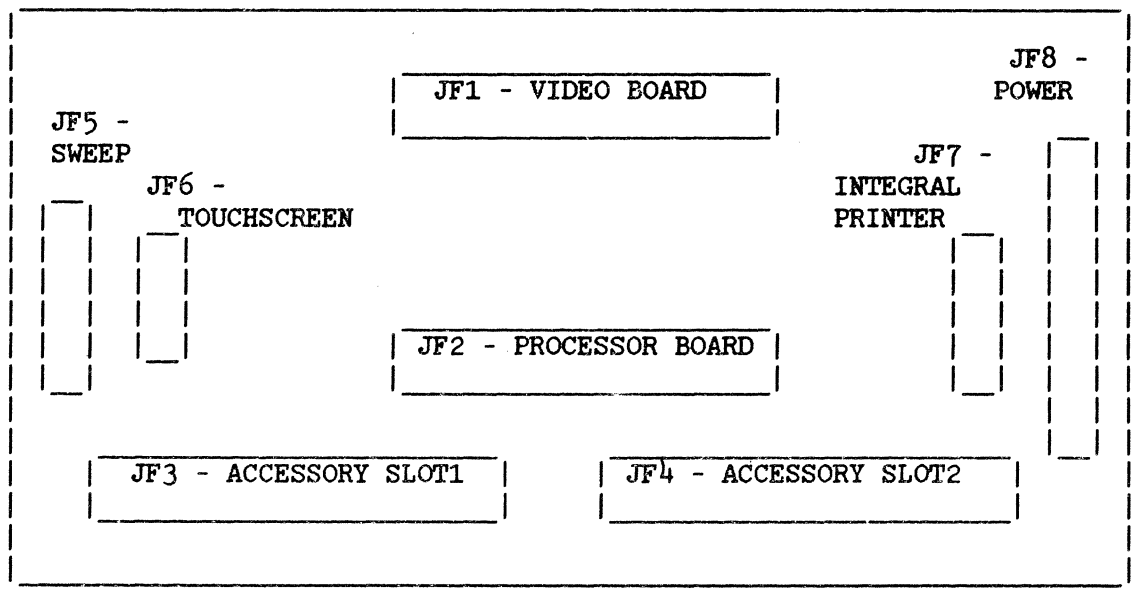
The TPM connects via a ribbon cable to a connector on the front plane. The 8088 accesses the TPM at I/O address XX30 as described in the Memory and I/O Mapping section of this manual. Note that "XX" can be any (address) value.

Summarizing, a byte of data is sent to the TPM by writing the data at port XX30. The TPM acknowledges the receipt of the data and readiness to accept the next byte of data to be sent by asserting the NOCINT interrupt line. The interrupt handling routine can poll the TPM by reading port XX30 and inspecting the least significant bit. If the bit is set, the TPM has generated an interrupt. The end of the read cycle clears the interrupt if caused by the TPM. PAPER OUT and ONLINE status are also available at the same address.

Front Plane Connectors and Signals

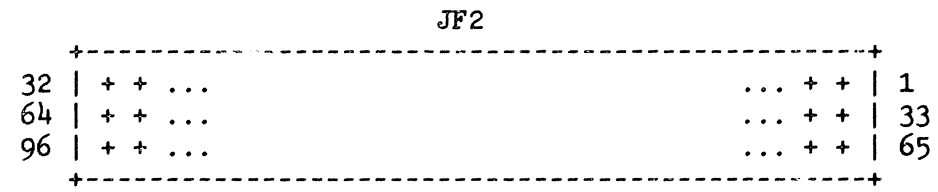
The following paragraphs provide a description of the connectors and the connector signals on the front plane.

Front Plane PCA Connector Layout:



JF2 - Processor Board Front Plane Connector Pinouts and Signals

1.	FPA 0	33.	NTPMWRT	65.	NTPMRD
2.	FPA 1	34.	NPFAIL	66.	GND
3.	FPA 2	35.	NSLOTSEL 2	67.	GND
4.	FPA 3	36.	FPNRST	68.	+5V
5.	FPA 4	37.	NOCINT	69.	GND
6.	FPA 5	38.	NOCWAIT	70.	+5V
7.	FPA 6	39.	+5V	71.	GND
8.	FPA 7	40.	BIO/-M	72.	GND
9.	ABUS 8	41.	-12V	73.	GND
10.	ABUS 9	42.	NVIDINT	74.	GND
11.	ABUS 10	43.	-12V	75.	GND
12.	ABUS 11	44.	FPGO	76.	GND
13.	FPA 12	45.	GND	77.	GND
14.	FPA 13	46.	GND	78.	GND
15.	FPA 14	47.	FPCLK	79.	GND
16.	FPA 15	48.	GND	80.	GND
17.	FPA 16	49.	GND	81.	GND
18.	FPA 17	50.	BIO/-M	82.	GND
19.	FPA 18	51.	+12V	83.	GND
20.	FPA 19	52.	FPDT/-R	84.	GND
21.	FPD 0	53.	+12V	85.	GND
22.	FPD 1	54.	SHOLDA	86.	GND
23.	FPD 2	55.	FPNWRT	87.	GND
24.	FPD 3	56.	N.C.	88.	+5V
25.	FPD 4	57.	FPNRD	89.	+5V
26.	FPD 5	58.	+5V	90.	+5V
27.	FPD 6	59.	N.C.	91.	+5V
28.	FPD 7	60.	GND	92.	TSDATA
29.	FPNSS0	61.	TSCLK	93.	GND
30.	NSLOTSEL1	62.	GND	94.	NDCOCINT
31.	BATV	63.	N.C.	95.	TSSYNC
32.	BATV	64.	FULLMEM	96.	+5V

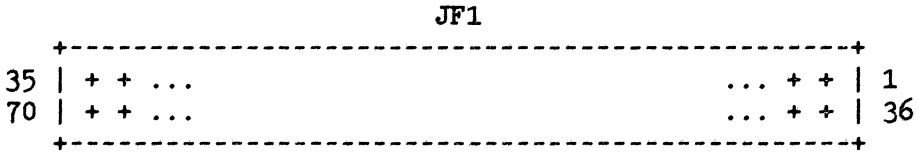


Pictorial view of processor board front plane connector with pin assignment (as viewed from component side of front plane board)

Video Board Front Plane Connector

JF1 - Video Front Plane Connector Pinouts and Signals

1.	FPA 0	36.	+5V
2.	FPA 1	37.	GND
3.	FPA 2	38.	FPNRST
4.	FPA 3	39.	GND
5.	FPA 4	40.	NOCWAIT
6.	FPA 5	41.	+5V
7.	FPA 6	42.	BIO/-M
8.	FPA 7	43.	GND
9.	ABUS 8	44.	NVIDINT
10.	ABUS 9	45.	GND
11.	ABUS 10	46.	FPGO
12.	ABUS 11	47.	GND
13.	FPA 12	48.	FPCLK
14.	FPA 13	49.	GND
15.	FPA 14	50.	+5V
16.	FPA 15	51.	GND
17.	FPA 16	52.	BIO/-M
18.	FPA 17	53.	GND
19.	FPA 18	54.	FPDT/-R
20.	FPA 19	55.	N.C.
21.	FPD 0	56.	+5V
22.	FPD 1	57.	N.C.
23.	FPD 2	58.	FPNRD
24.	FPD 3	59.	GND
25.	FPD 4	60.	(RESERVED)
26.	FPD 5	61.	GND
27.	FPD 6	62.	+5V
28.	FPD 7	63.	GND
29.	+5V	64.	NVSYNC
30.	+5V	65.	GND
31.	+12V	66.	NHSYNC
32.	+12V	67.	GND
33.	+12V	68.	NHB
34.	BATV	69.	NFB
35.	BATV	70.	GND



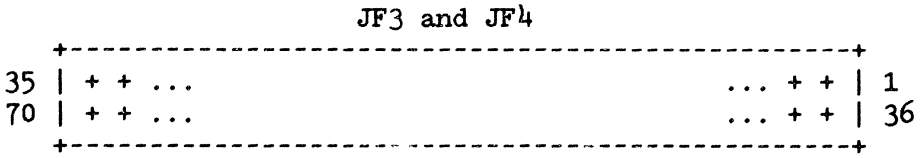
Pictorial view of processor board front plane connector with pin assignment (as viewed from component side of front plane board)

Accessory Front Plane Connectors

The accessory boards plug into the bottom slots of the card cage. The system signals available to the modules are listed below.

JF3/4 Accessory Board Front Plane Connector Pinout and Signals

1.	FPA 0	36.	(RESERVED) *
2.	FPA 1	37.	NPFAIL
3.	FPA 2	38.	(RESERVED) *
4.	FPA 3	39.	FPNRST
5.	FPA 4	40.	+5V
6.	FPA 5	41.	GND
7.	FPA 6	42.	GND
8.	FPA 7	43.	NOCINT
9.	ABUS 8	44.	GND
10.	ABUS 9	45.	NOCWAIT
11.	ABUS 10	46.	(RESERVED) *
12.	ABUS 11	47.	BIO/-M
13.	FPA 12	48.	+5V
14.	FPA 13	49.	-12V
15.	FPA 14	50.	NDCOCINT
16.	FPA 15	51.	GND
17.	FPA 16	52.	-12V
18.	FPA 17	53.	SHOLDA
19.	FPA 18	54.	FPGO
20.	FPA 19	55.	+5V
21.	GND	56.	GND
22.	GND	57.	FPCLK
23.	FPD 0	58.	(RESERVED) *
24.	FPD 1	59.	GND / FULLMEM *
25.	FPD 2	60.	+12V
26.	FPD 3	61.	GND
27.	FPD 4	62.	BIO/-M
28.	FPD 5	63.	+12V
29.	FPD 6	64.	FPNWRT
30.	FPD 7	65.	+5V
31.	GND	66.	FPDT/-R
32.	FPNRD	67.	GND
33.	GND	68.	FPNSSO
34.	+5V	69.	GND
35.	BATV	70.	NSLOTSELx **



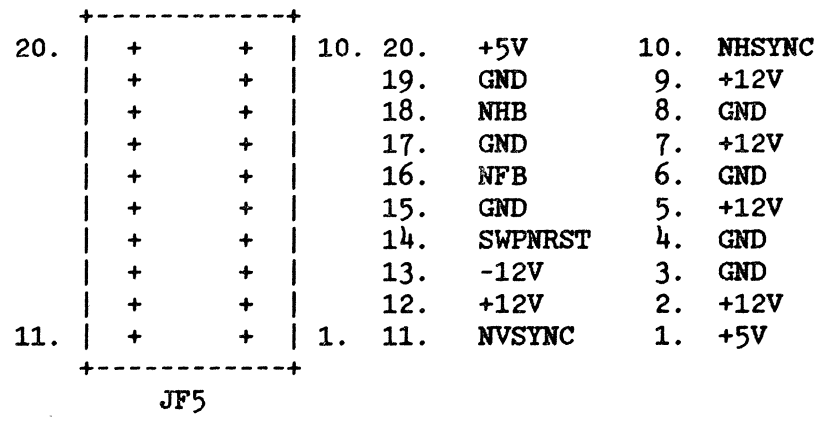
Pictorial view of option module front plane connector with pin assignment (as viewed from component side of front plane board)

* Pin 59 is FULLMEM is on JF3 only. Pin 59 is GND on JF4. Connections to this pin and all pins labeled RESERVED should not be made.

** NSLOTSELx will be either NSLOTSEL1 or NSLOTSEL2 depending upon which side of the front plane the module is plugged into. The left side of the card cage (from rear view of the package) holds module 1 and gets the NSLOTSEL1 signal. The right side of the card cage holds module 2 and gets NSLOTSEL2.

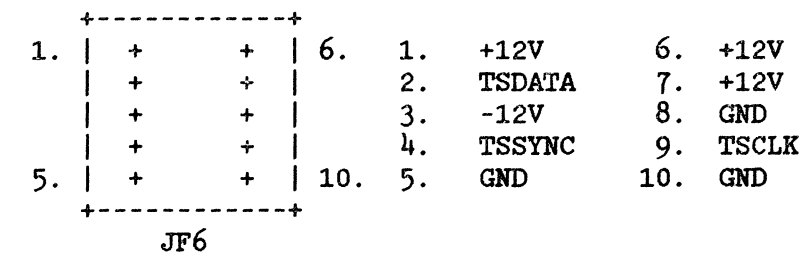
Sweep Board Connector

The Sweep board connects to the front plane via a 20 pin connector, JF5, whose signal pins are listed below:



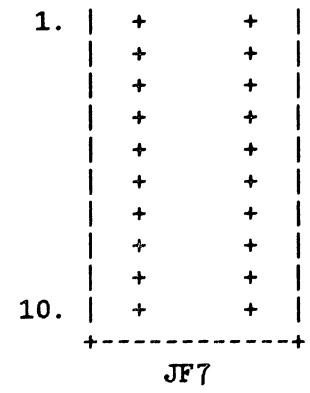
Touchscreen Connector

The Touchscreen connects to the front plane via a 10 pin connector, JF6, whose signal pins are listed below:



Integral Printer Connector

The Integral Printer connects to the front plane via a 20 pin connector, JF7, whose signal pins are listed below.

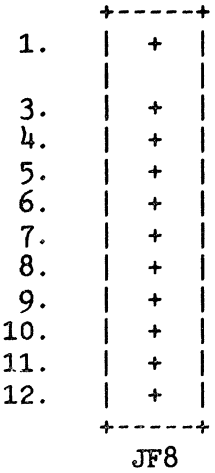


	1.		11.	1.	D0	11.	D1
				2.	D2	12.	D3
				3.	D4	13.	D5
				4.	D6	14.	D7
				5.	NPOR	15.	NSTR
				6.	NACK	16.	POUT
				7.	N.C.	17.	NONL
				8.	GND	18.	GND
				9.	+5V	19.	+5V
	10.		20.	10.	N.C.	20.	N.C.

JF7

Power Supply Connector

Power to the system comes from the power supply through a 12 pin connector, JF8, on the front plane board. The signal definition is given below:



1.	NPFAIL
	(Pin removed for polarization)
3.	GND
4.	-12V
5.	GND
6.	(N.C.)
7.	+12V
8.	GND
9.	+5V
10.	+5V
11.	+5V
12.	GND

JF8

Front Plane Signal Descriptions

The following description of the front plane signals reference the connector diagrams shown previously.

FPA 0-7 — The lower 8 address bits of the 20 bit address generated by the 8088. These signals are buffered and demultiplexed.

ABUS 8-11 — The upper 12 address bits of the 20 bit address.

FPA 12-19 — These signals also are buffered and demultiplexed.

FPDBUS 0-7 — Data bus signals from the external data bus. This data path is the means through which data is passed between the processor and the video board or option slot modules or the optional TPM.

FPCLK — Buffered 8 Mhz system clock.

FPNRD — Buffered -RD signal from the 8088 used to indicate a bus read cycle in progress.

FPNWRT — Buffered -WRT signal from the 8088 used to indicate a bus write cycle in progress.

FPNSS0 — 8088 bus cycle status line. The combination of FPNSS0, BIO/-M, and FPDT/-R allow boards connected to the front plane to completely decode the current bus cycle.

BIO/-M — Buffered IO/-M signal from the 8088 used to distinguish memory and I/O bus cycles.

FPDT/-R — Buffered DT/-R signal from the 8088 used to indicate direction of data from the 8088 for a given bus cycle.

FPGO — Signal which qualifies the address generated by the microprocessor. The address qualification is needed primarily for dynamic RAM circuits which cannot tolerate an assertion of RAS or CAS on a false address. FPGO goes high at the beginning of T2 and goes back low at the beginning of T4 of a bus cycle. FPGO can also be used to terminate a bus write cycle by having its falling edge used to clock data into a register or other device on an accessory device in an option slot. Terminating the writes in this way can provide better hold timing than using FPNWRT.

NOCWAIT — This line can be asserted by the option modules or video board through an open collector gate to insert wait states into a bus cycle to provide sufficient time for a bus cycle access.

NVIDINT — This line is asserted by the video subsystem to generate an interrupt to the processor. This interrupt occurs once every frame. The signal can be monitored by the option slot devices if desired.

NDCOCINT — This open collector interrupt signal has the same interrupt priority as the datacomm controller chip on the processor board as it shares the same input to the interrupt controller as the datacomm controller chip does. This input can be asserted by accessories for interrupt servicing.

NOCINT — System interrupt signal asserted by a device via an open collector gate. This line can be used by options slots to get processor service.

SHOLDA	Synchronized hold acknowledge signal from the 8088. HOLD is asserted by the mezzanine memory board.
FULLMEM	Indicates which mezzanine memory board option is installed. FULLMEM=0 if 128K RAM and FULLMEM=1 (it is pulled high by a pull up resistor on an installed extension memory board) if 256K RAM is on the board. An accessory should not connect to this signal pin.
NSLOTSEL1	Signal indicating an address within the 64K block allocated to option module 1 has been generated by the 8088.
NSLOTSEL2	Same as NSLOTSEL1 but asserted when address within option module 2 address space is generated.
NFB	Video Full-Bright signal to sweep board.
NHB	Video Half-Bright signal to sweep board.
NHSYNC	Horizontal sync signal from video board used by the processor and sweep boards.
NVSYNC	Vertical sync signal from video to sweep board.
TSCLK	Touchscreen clock signal.
TSDATA	Touchscreen data signal.
TSSYNC	Touchscreen sync signal.
NTPMRD	Signal from CPU to read TPM status.
NTPMWRT	Signal generated by the processor card to handshake bytes of data to the TPM.
FPNRST	Signal generated by processor board at power-on time to initialize logic circuitry.
NPFAIL	Signal generated by the power supply indicating power supply output level stability.
BATV	These two connector lines carry current from the batteries located on the video board to circuitry requiring battery back-up power such as the CMOS RAM and the CMOS real time clock.

PROCESSOR SUBSYSTEM

Overview

The HP 150 processor board is based on the Intel 8088-2 microprocessor. The microprocessor runs at an 8 MHz clock speed and is configured for operation in the minimum mode. The general architecture of the processor, memory, and I/O is depicted in figure 3-1. As shown in the diagram, the majority of the circuitry is partitioned between three data busses: the memory data bus, the I/O data bus, and the external data bus.

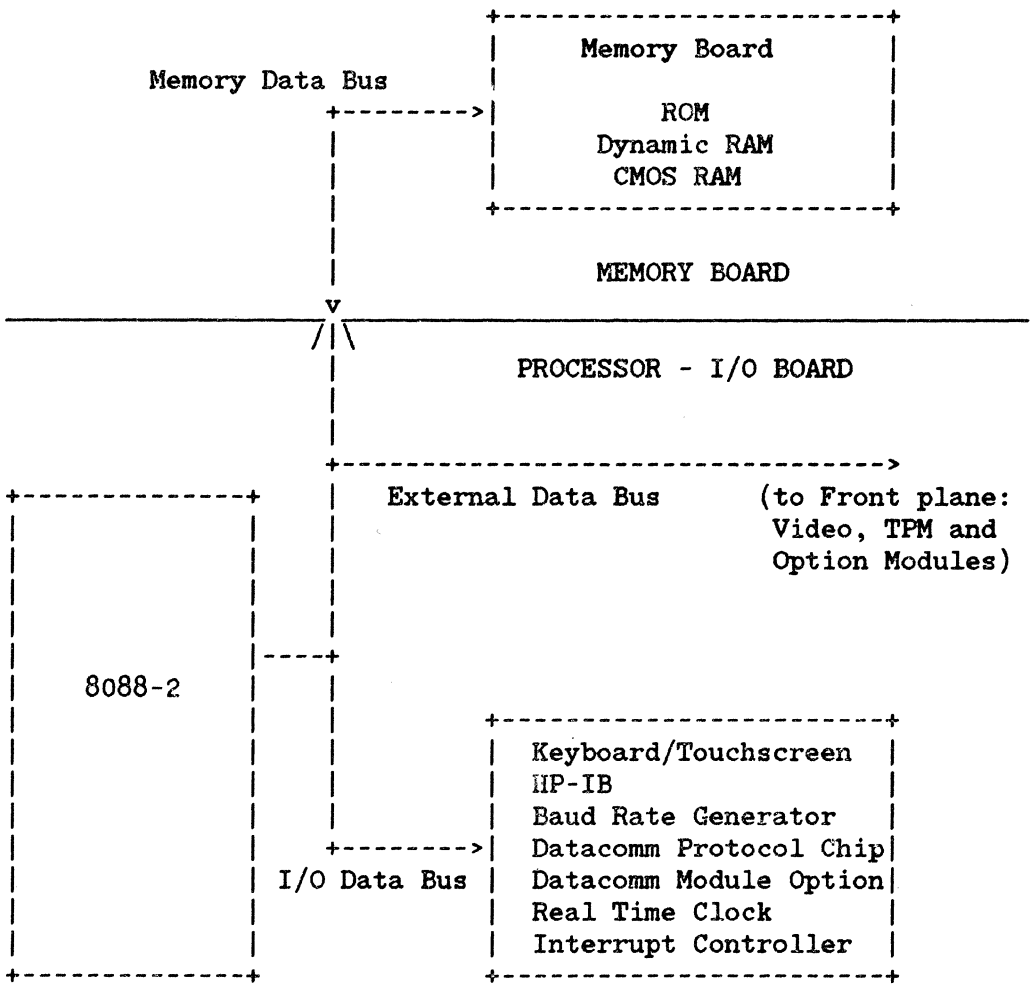


Figure 3-1. Processor Board and Memory Board Block Diagram

Memory and I/O Mappings

The logic on the processor board and the other boards which exchange data with the microprocessor are mapped into the 8088's memory or I/O space. Section 4 depicts the memory and I/O maps for the system. Note that I/O addresses XX80 through XXFF are available for option module use. Also note that "XX" can be any value.

Processor Board Block Diagram

Besides the 8088 microprocessor, the processor board contains a number of LSI and MSI components which are used to interface with other subsystems, provide communications with external devices, or perform specialized functions. The block diagram in figure 3-2 shows the basic architecture of the board. This section of the manual will detail the various subsections and components of the processor board.

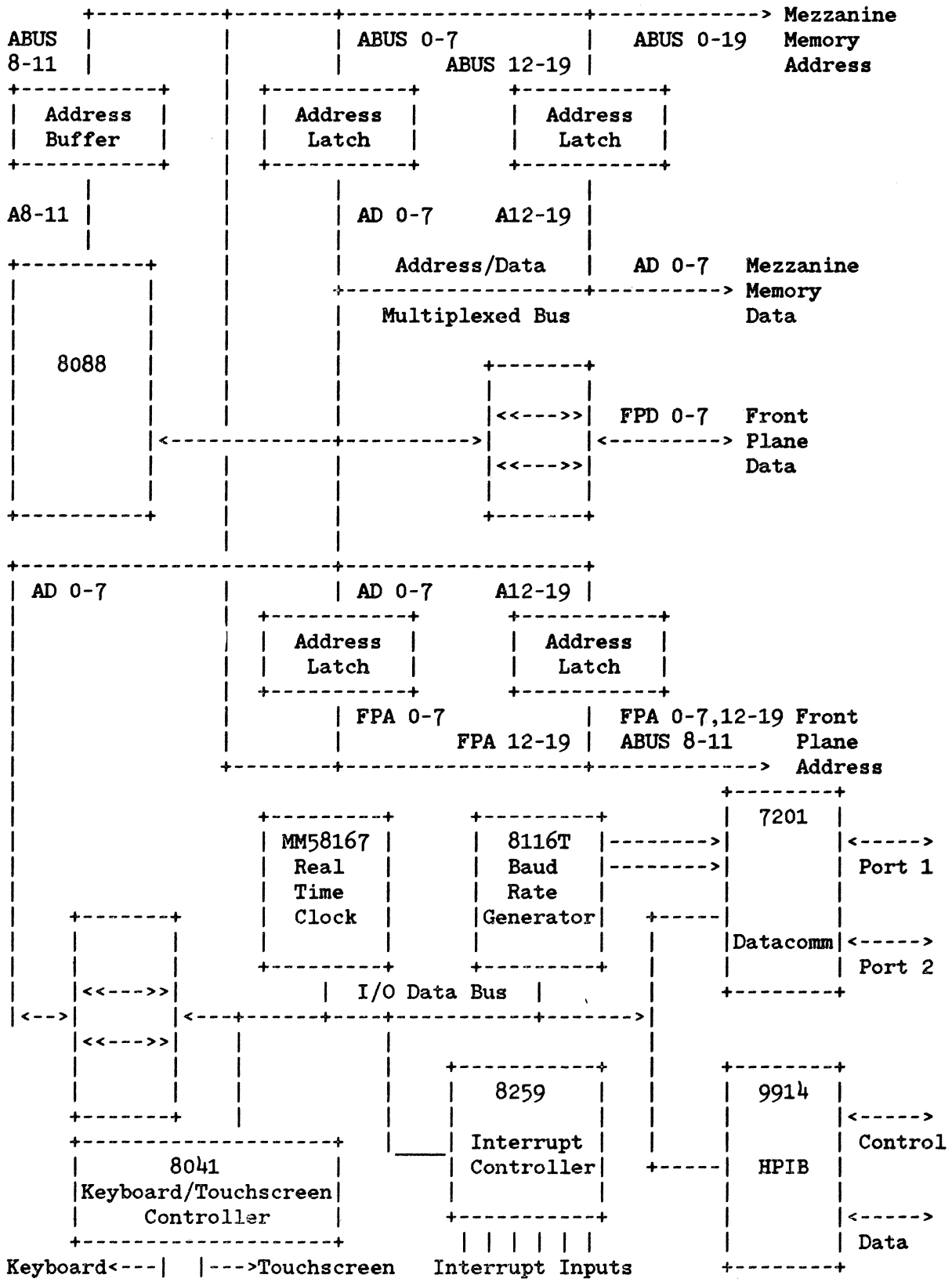


Figure 3-2. Processor Board Block Diagram

Reset, CMOS Power, and Test Strap Logic

This set of miscellaneous circuits provide functions of an ancillary or support nature to the more visible functions of the board to be described later.

RESET LOGIC. The power-fail signal from the power supply (NPFAIL, J1-34) is used by the system as a power-on-reset and a CMOS device standby operation mode switch. NPFAIL is low when power is first applied to the system and goes high no sooner than 75 msec after the power supply output voltages have stablized at their specified values. When NPFAIL goes low due to the power being disconnected from the power supply, it does so at least 500 usec before the power supply output voltages go out of regulation.

CMOS POWER CIRCUIT The CMOS power circuit regulates current use by real time clock and configuration CMOS memory between the batteries and the power supply while the unit is switched on.

TEST STRAP LOGIC. If the test strap input is held low during execution of manufacturing self-test, a failing test will be iterated until the failure ceases or the input is brought high. This is intended to be a convenience for servicing purposes.

Microprocessor System Architecture

The system microprocessor is an 8088. The clock frequency the 8088 is run at is 8 Mhz. The 8088 is configured in the "minimum mode." As it is not the intent of this document to review microprocessor basics, please refer to the Intel 8088 data sheets and application notes for fundamentals on 8088 operation.

BUS CYCLES

Address Generation. At the beginning of a bus cycle (memory or I/O access), the 8088 places the memory or I/O address on its address and address/data pins. The address is multiplexed with the data and some of the status information so the processor board uses transparent latches to latch and buffer the address generated at the beginning of the cycle. Address bits A8-A11 are not multiplexed so a non-latching buffer is used for buffering those four signals. ALE (U211 pin 25) is the signal used to latch the address into the latches.

Data Transactions. The 8088 does all data transactions through one of three bus tranceivers. All the mezzanine memory board components are isolated by a transceiver located on the memory board. All the I/O devices (real time clock, datacomm, baud rate generator, keyboard/touchscreen controller, HP-IB controller, etc.) are on the I/O bus which is separated from the 8088 by another transceiver on the processor board. Any access not to the mezzanine memory or the I/O bus will default to the front plane.

Status Generation. The 8088 generates status information indicating the type of bus cycle in progress. Control signals based upon these are generated by the processor and are buffered and sent throughout the system to allow circuitry to respond properly to read, write, memory, or I/O cycles.

HOLD State. The 8088 will be put into a HOLD state by the mezzanine memory approximately every 56 usec during dynamic RAM refresh.

System Timing and Control Logic

CLOCK GENERATOR. The 8284A clock generator is used to provide the system clock for the HP 150 hardware. A 24 Mhz crystal provides the reference frequency used by the 8284. The 8284 creates an 8 MHz clock which nominally is high 33% and low 66% of the 125 nsec period. The 8284 also divides the 8 Mhz clock by 2 to form the 4 Mhz signal used as the system clock for the 7201 datacomm controller, the 8116T baud rate generator, the 9914A HP-IB controller, and the 8041A keyboard and touchscreen controller.

The 8284 also provides a reset output pin which is used as a basis for the reset signals throughout the system hardware.

The -F/C input (pin 13) of the 8284 is at a logic 0 state which selects the crystal oscillator as the source frequency for the 8284.

The 8284 is a part of the wait state insertion logic. The 8284 is configured in the synchronous mode which affects some of the timing constraints the wait state generator logic has. Wait state generation is described in the next section.

WAIT STATE GENERATION. The basic bus cycle for the 8088 consists of four 125 nsec clock periods during which the memory of I/O address is generated by the CPU followed by a transmission of data to or the reception of data from a memory or I/O device. The cycle can be extended by adding "wait states" to the basic 4 clock cycle. The standard bus cycle is shown in figure 3-3.

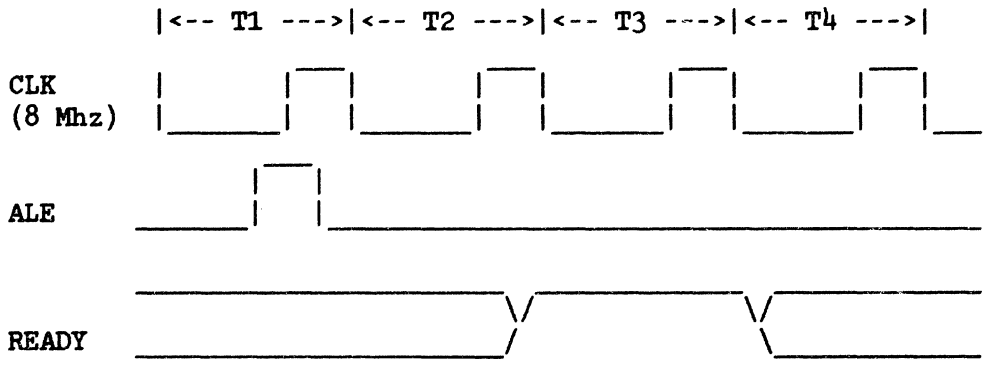


Figure 3-3. Standard Bus Cycle with no Wait States

The four clock periods are labeled T1 through T4 for reference purposes. The 8088 samples the READY input during the low clock period of T3. If it is high at that time, no extra wait states are added. If it is low, a wait state is added as shown in figure 3-4. READY is sampled during the low clock period of every wait state. Whenever it is sampled high, the wait state is exited and the bus cycle goes into the T4 state.

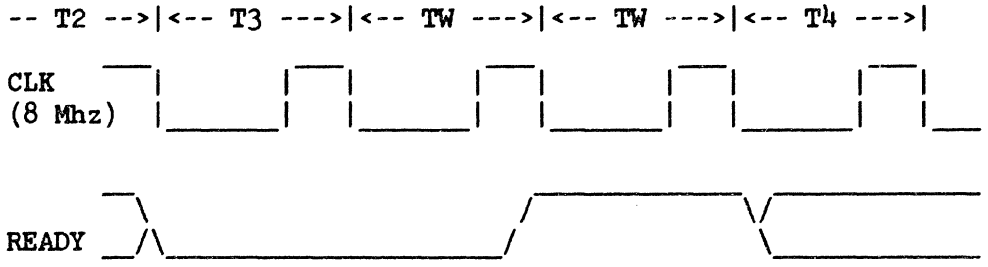


Figure 3-4. Bus Cycle with Two Wait States

The READY input to the 8088 is driven by the RDY output (pin 5) of the 8284. The 8284 has two inputs, RDY1 and RDY2 (pins 4 and 6), by which wait states are controlled. Normally, wait states are added by RDY1 (NWAIT). RDY1 is sampled at the falling edge of each clock, so effectively, the READY input of the 8088 is controlled by what the 8284 samples at its RDY1 input just prior to the beginning of T3 or just prior to the beginning of each wait state.

Some processor board logic exists to add a specific number of wait states to certain bus cycles. Two wait states are added to the bus cycle when the 8814 or 8041 or 8116 are accessed by the 8088 and 9 wait states are added when the real-time clock is accessed.

A minimum of one wait state is added to every bus cycle when jumper W1 is installed. (Access of devices requiring more wait states get the additional wait states needed but one wait state appears in bus cycles on accesses of devices whose decoding logic isn't tied into the wait state generator.) Installing W2 causes no wait states to be added for ROM access only.

Wait states may be inserted by other logic boards by asserting the NOCWAIT line that is routed to the front plane option slots and video board connector. As long as NOCWAIT is low, wait states will be executed by the 8088. Since the NOCWAIT input goes into a synchronizing flip-flop the wait states added by an option or the video board must be anticipated in a timely fashion. The timing diagram in figure 3-5 illustrates wait state insertion and deletion by an option card.

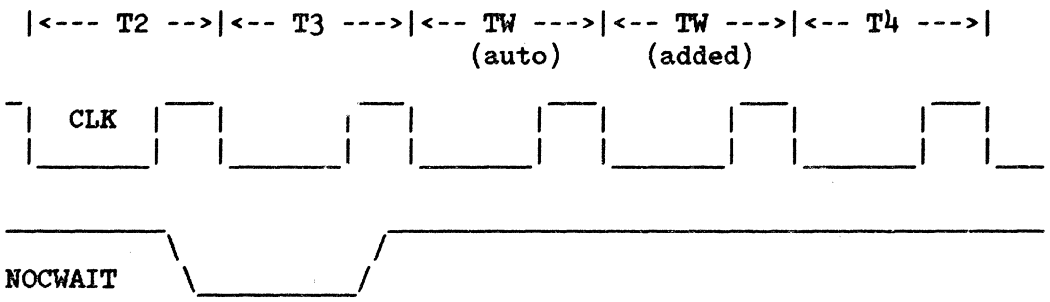


Figure 3-5. Option Wait State Insertion

To put in an additional wait state (there is one automatic wait state for all over the front plane bus cycles) the option must assert NOCWAIT no later than 2 clocks prior to the time the wait state is desired. Wait states are exited 2 clocks after the next falling edge of CLK from the point where NOCWAIT is deasserted. If an option is designed whose enable is qualified by FPGO (described in the next section) wishes to put in a wait state, it must assert NOCWAIT within: Tclk - Tfpgo - Tsu74 - 2Tpfp = 125- 38-3-20 = 64 nsec of the falling edge or beginning of T3. (Tclk = clock period, 125 nsec; Tfpgo = front plane GO delay; Tsu74 = 74S74 data setup time, Tpfp = front plane propagation delay.)

GO GENERATOR. The purpose of the GO generator is to provide a signal to qualify the address generated by the 8088 on its address pins. The GO generator indicates the beginning of T2, end of T1 within a bus cycle and the beginning of T4. The address qualification is needed primarily for dynamic RAM circuits which cannot tolerate an assertion of RAS or CAS on a false address. Another feature of the GO signal is that its deassertion is useful for providing better hold margin timing on bus write cycles. A state machine is used to implement the GO generator. Basically, the machine inspects the NWAIT signal and ALELCH. When ALELCH goes high, the GO generator brings GO high at the next negative edge of the clock (beginning of T2). NWAIT is monitored until NWAIT goes high. Two clocks later, GO is brought low (at the beginning of T4) indicating the end of the bus cycle.

The machine algorithm is:

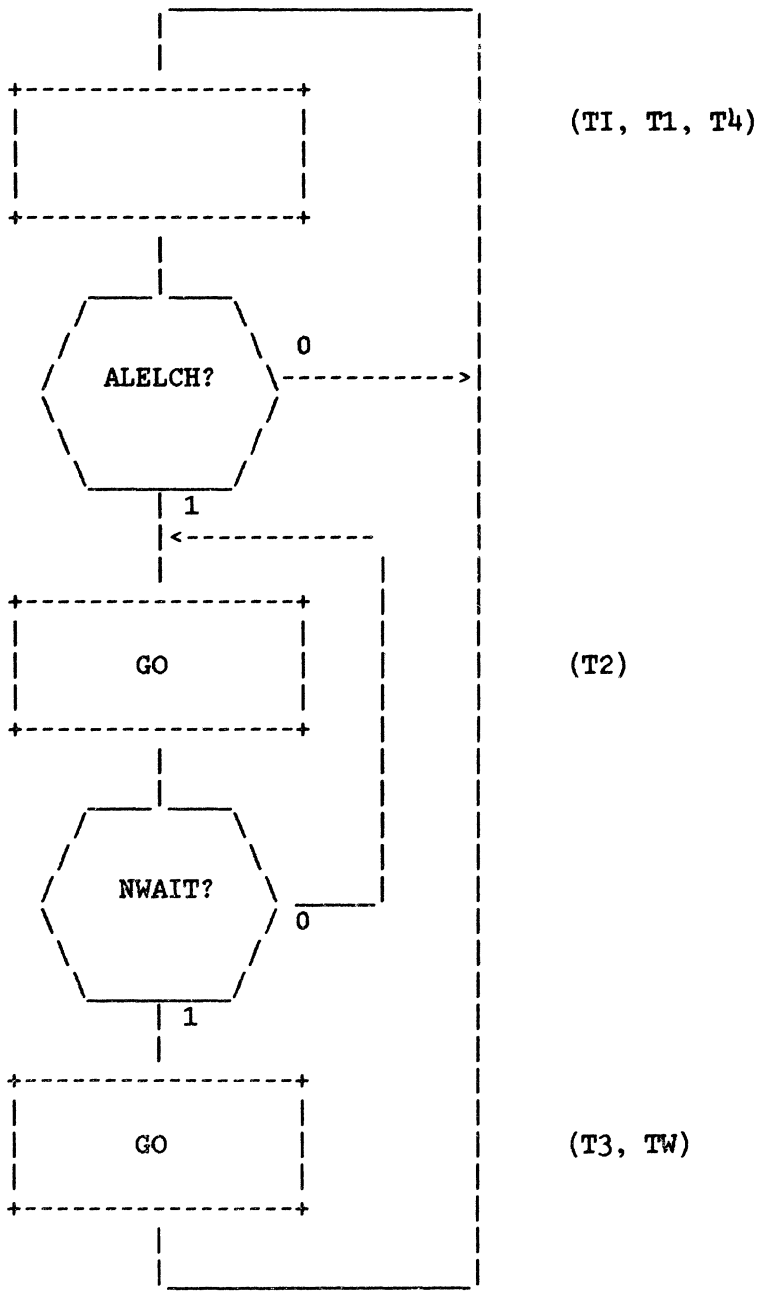


Figure 3-6. GO State Machine Algorithm

The HP 150 utilizes an 8284 clock generator (READY generator) operating in synchronous mode. Therfore, the RDY1 input of the 8284 (pin 4) must be stable 35 nsec before each falling edge of the clock. The GO generator circuit requires NWAIT to be stable 20 nsec prior to the falling edge. Thus, meeting the 8284 specification meets the GO generator NWAIT timing requirement.

The timing relationship between GO and the other system timing is portrayed in figures 3-7 and 3-8.

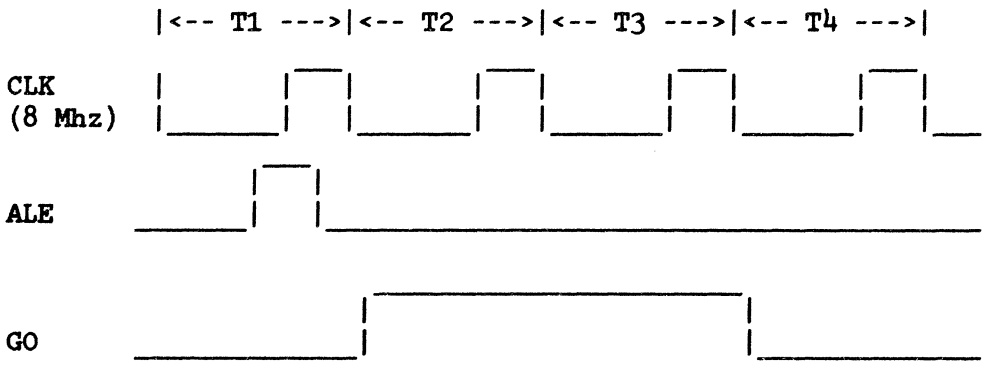


Figure 3-7. Standard Bus Cycle with no Wait States with GO Timing.

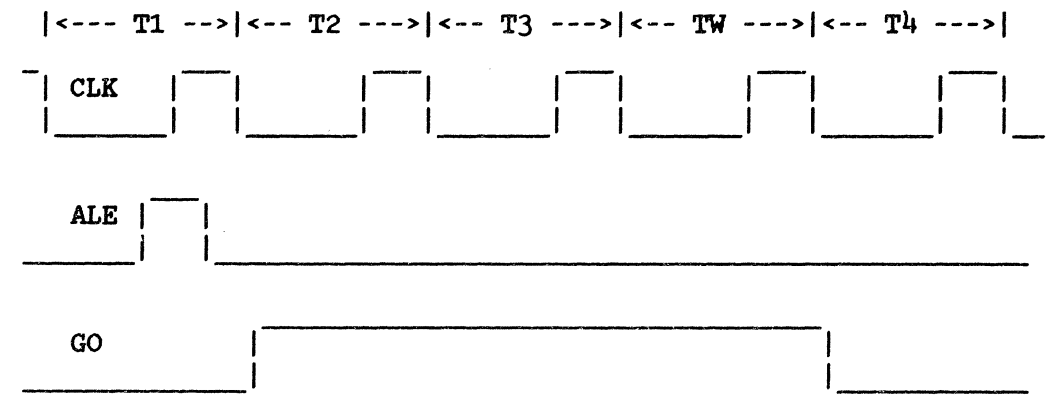


Figure 3-8. Bus Cycle with One Wait State With GO Timing.

The skew between GO and the falling edge of the clock should not exceed 20 nsec.

Interrupt Controller

The 8088 is capable of processing two types of hardware interrupts. One type is the non-maskable interrupt (NMI) which is processed when the NMI input to the 8088 makes a low to high transistion. This occurs when a hard reset key sequence (CTRL-SHIFT-RESET) is issued by the user. The 8088, after getting an NMI, will use the values in memory locations 8,9,A,B for new code segment and instruction pointer values. This will cause the 8088 to vector to an interrupt service routine.

The other type of interrupt processed by the 8088 is standard or maskable interrupts. This type of interrupt is invoked by the INTR input of the 8088 going high. In response the 8088 will lower the NINTA or interrupt acknowledge output twice. The second time NINTA goes low, the 8259A interrupt controller will place an 8 bit vector on the data bus. This vector will be a value from 0 - 0FFH. The 8088 will start at the memory location equal to 4 times the vector and load the content of that location and the next three locations into the instruction pointer and code segment registers and subsequently branch to the interrupt routine.

The 8259 has 8 inputs through which it accepts incoming interrupts from various sources. When an interrupt request occurs by one or more of the inputs IR0 - IR7 going high, it raises the INTR input to the 8088 high. The 8088 in turn asserts NINTA two times. The 8259 will select one of the eight interrupt input sources for processing and when the 8088 sends the second pulse on NINTA, the 8259 will place a vector on the bus that corressponds to the interrupting device that is to be serviced. Should several interrupt requests come into the 8259 simultaneously, the 8259 chooses which one is to be processed first and which ones subsequently.

The interrupts coming into the 8259 are prioritized as:

Highest priority	IR0	Video
	IR1	Datacomm (7201 and front plane interrupts)
	IR2	+5V (Video second level tasks)
	IR3	Keyboard and Touch Screen
	IR4	Front plane interrupts (lower level)
	IR5	HP-IB
	IR6	(not used)
Lowest priority	IR7	Real-Time Clock

The 8259 can be programmed to handle incoming interrupts in a number of different ways. The system firmware puts it in the iAPX 86 mode, level-triggered and non-buffered modes. The 8088 programs the 8259 by writting various data bytes to it over the I/O bus. The 8259 can be programmed to ignore certain interrupts, handle incoming interrupts in a particular order, or even to work as a polling device if desired.

The various ways the 8259 can be used are described in the 8259 data sheet and application note available from Intel.

I/O Devices

The I/O bus has eight functional modules which exchange data with the microprocessor. These modules are the 8259A interrupt controller, the MM58167A real-time clock, the 8116T baud rate generator, the 8041A microcomputer which controls the keyboard and touchscreen, the 9914A HP-IB controller, 7201 datacomm controller, and logic pertaining to datacomm on the fixed port (port 2) as well as the mezzanine datacomm board (port 1). Figure 3-9 illustrates the I/O bus modules.

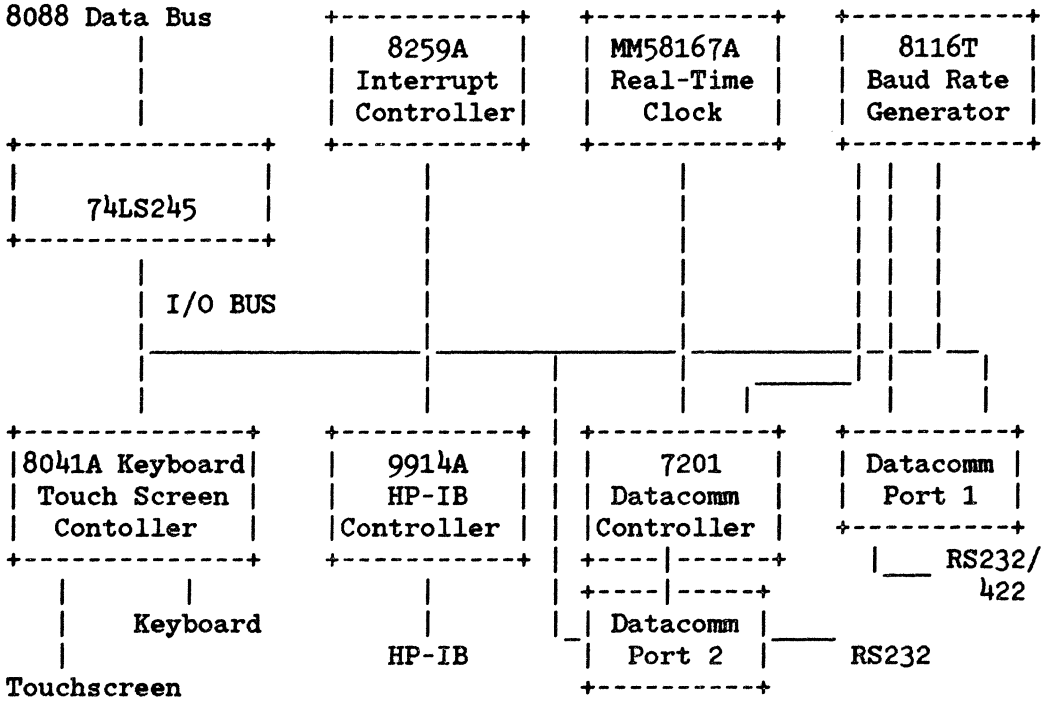


Figure 3-9. I/O Bus Device Block Diagram

I/O DECODING. One or more I/O ports are assigned to each device or module on the I/O bus to provide the required communication path with the CPU. The I/O map (presented in Section 4) lists the port addresses assigned to each device. An I/O bus cycle (one that is not an interrupt acknowledge cycle) is executed by the 8088 with the IO/-M signal high, the NSSO signal high and the DT/-R signal low, or with IO/-M high, NSSO low and DT/-R high. During the I/O cycle the lower 16 bits of the address bus carry the port address being accessed. An OR and a NAND gate monitor BIO/-M, NSSO, and DT/-R to see if an I/O cycle is occuring. A 3-line-to-8-line decoder with a NOR gate decode the address on the address bus and enable the 7201, the 9914, the 8041, the 8259, the 8116, or logic in one of the datacomm ports when an I/O bus cycle is issued by the 8088 to access one of these devices. A NAND gate, a NOR and an inverter are used to decode the addresses used by the real-time clock chip.

KEYBOARD AND TOUCHSCREEN CONTROLLER. The interface between the 8088 and the keyboard is provided by a single chip microcomputer, an Intel 8041. Additionally, the 8041 provides the interface to the touchscreen.

The 8041 interfaces to the 8088 over the data bus. Two I/O ports are devoted to it, ports 18H and 19H. The 8088 can send various commands to the 8041 while the 8041 can send information such as a key number or a touchscreen report.

Keyboard: The interface between the 8041 and the keyboard depends on the synchronization of a hardware counter on the keyboard, and a software counter in the 8041. With each key clock the 8041 sends to the keyboard, it increments the software counter. It then checks the data line for status of the currently addresed key. The contents of the software counter specify the correct key address.

The KBDSYNC line provides a reset mechanism allowing the 8041 to put the keyboard counter into a known state (count = 0). Thus the interface to the keyboard is provided by three lines: KBDCLK from 8041, KBDSYNC from 8041, and KBDDATA from keyboard.

Touchscreen: The interface to the touchscreen is essentially identical to that for the keyboard. The most significant difference is that the touchscreen drives the TSSYNC line. The 8041 must send clocks until the touchscreen activates the TSSYNC line to determine the state of the touchscreen counter. This provides a positive indication as to whether the touchscreen is present. Again, the interface is over three lines: TSCLK from the 8041, TSSYNC from the touchscreen, and TSDATA from the touchscreen.

Bell: The 8041 also provides the drive for the system bell. The on-board timer circuit is used to select the frequency of the bell tone.

Circuitry between the 8041 and the keyboard converts the TTL levels from the 8041 to 12 volt reference levels used on the keyboard. This is also true for the touchscreen.

For a more detailed explanation of how the 8041 interfaces to the keyboard and touchscreen see "8041 Keyboard/Touchscreen Scanner", which is discussed later in this manual.

DATACOMM. The HP 150 hardware provides two independent datacomm ports. These ports provide a full complement of signals and a large degree of flexibility for the system firmware to take advantage of. A description of these datacomm facilities is discussed later in this manual under "Datacomm Subsystem".

HP-IB CONTROLLER AND INTERFACE. The HP-IB controller used in HP 150 is the TMS9914A which is capable of handling the talker and listener functions, parallel and serial poll functions, and data handshaking functions required to implement the HP-IB standard protocol. SN75160 and SN75161 bus transceivers are used to interface the controller to the HP-IB bus.

The 9914 system clock is the 4 Mhz square wave generated by the 8284 (PCLK). The 9914 has 8 I/O addresses assigned to it by which the CPU accesses the controller's internal 14 registers to do HP-IP transactions.

The 9914 can also generate an interrupt to the system at interrupt level 5 to receive CPU attention if the processor has enabled the interrupt facilities. -INT, the interrupt output signal is an open-drain active low signal which is pulled up and inverted for 8259 interrupt handling.

REAL TIME CLOCK. The HP 150 uses the MM58167 CMOS real time clock (RTC). This is a battery backed up component as is the CMOS RAM on the mezzanine memory board. When the main system power is turned off, the 3V battery and the associated circuitry maintains approximately 2.5V at the power supply pins of the CMOS parts.

The '58167 is a versatile chip. It interfaces to the system with an 8 bit bus. The RTC keeps track of the month, day of month, and day of the week as well as the time in hours, minutes, seconds, tenth seconds, hundreths and milliseconds. It is capable of producing repeated interrupts at the rates of 1/month, 1/week, 1/day, 1/hour, 1/minute, 1/second and 10/second. An alarm feature allows an interrupt to be sent to the system at a designated time.

Several components external to the chip are used in the oscillator circuit for the RTC. These include a variable capacitor which must be tuned for accurate time keeping with this circuit. The suggested way to calibrate the variable capacitor is to connect a frequency counter to the interrupt output of the RTC and program the chip to produce a repetitive 1 Hz interrupt. Adjust C28, the variable capacitor, until the period of the pulse is as close to 1 second as it will adjust to.

A number of components are used to create the final signals used to drive the -RD and -WR inputs of the RTC. The '58167 has some rather unusual address and data setup and hold times that require the creation of special -RD and -WR signals to meet the RTC's timing constraints.

NOTE

The millisecond register of the RTC is not very accurate. It is not recommended for use in critical time applications.

The RTC interrupt status register cannot be polled! The status register must be interrogated after receiving an interrupt from the RTC. This is due to a race condition that exists in the chip that can cause an interrupt to be permanently masked if the status register is interrogated at the time the interrupt was about to occur. This implies that the RTC must not have its interrupt output shared in an open collector fashion with other devices unless it is always the last device to be polled in an interrupt service routine.

The RTC can continue to drive the data bus as long as 250 nsec after the -RD input is brought back high. A restriction that we have in the HP 150 system is that a read from the RTC cannot be followed immediately by a write to any device on the I/O bus.

VIDEO SUBSYSTEM

The HP 150 Video Board provides the alphanumeric and graphics displays. This section of the manual describes in detail the workings of the board. The video board interfaces to the rest of the system through the front plane board. The interface signals and connector pinouts are described earlier in this section.

Video Technology and Display Format

RASTER SCAN. The HP 150 uses a RASTER SCAN display technology in a non-interlaced mode at a 60Hz refresh rate. The video is generated by scanning the CRT (cathode ray tube) with an electron beam. Electrons striking the phosphorous coating on the inside of the tube cause it to glow. Characters are displayed using dot patterns traced out by the electron beam.

The HP 150's display system consists of:

1. Video Board
2. Sweep Board
3. CRT

The Video Board stores alphanumeric and graphics information in display RAMs. It then converts this information into a serial dot stream along with the necessary control signals and sends them to the Sweep Board.

The Sweep Board is responsible for controlling the CRT. It transforms the digital signals it receives from the Video Board into analog signals necessary for manipulating the electron beam.

The CRT acts as a transparent screen, onto which information is displayed. Electrons are generated in the back of the tube called the "electron gun." They are accelerated towards the front by the high voltage placed on the inside of the tube by the Sweep Board. The horizontal and vertical direction of the beam is regulated by a set of magnetic coils known as the "Yoke." The Yoke receives analog control signals from the Sweep Board also. See figure 3-10 for a block diagram description of the video system.

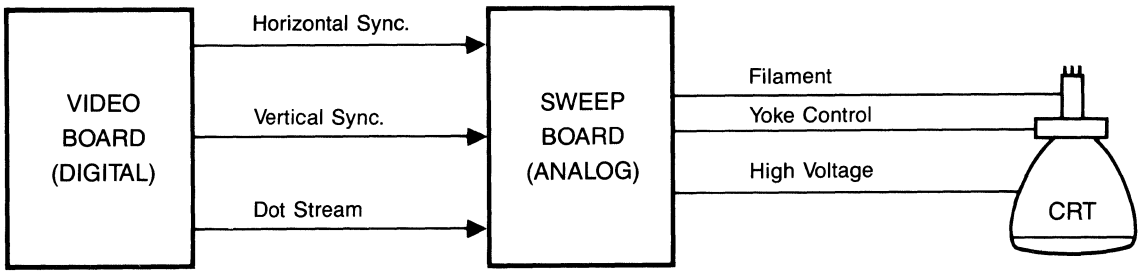


Figure 3-10. Video Subsystem Block Diagram

VIDEO FRAME FORMAT. Figure 3-11 shows the CRT pattern for the ALPHA display for one frame time. Part a) of the figure shows the begining of the frame when the electron beam is traced from the upper left portion of the screen to the lower right. Part b) shows the Vertical Retrace, when the beam is returned to the upper left of the screen, thus getting ready for the start of the next frame. For the sake of clarity, the following section refers to ALPHA display only. The relationship between Graphics and Alpha will be explained later.

Looking at figure 3-11, the following events take place during an ALPHA display in one frame time:

1. The beam starts out at the top of the tube in the "Start Scan Lines" area. The beam will not be allowed to turn on for the first 12 scan lines. This area is blanked so that the beam control signals coming from the Sweep Board have a chance to settle down before active video display.

2. On the 13th scan line, the beam reaches the active display area and video data is displayed. Note that horizontal retraces are necessary to bring the beam back to the start of the next scan line. During this time, however, the beam is turned off.

3. After all of the 378 ALPHA scan lines have been displayed we enter the 6 "Extra Scan Line" region, where the beam is blanked.

4. We reach the end of the "Extra Scan Lines" and the Sweep Board receives a Vertical Drive pulse, from the Video Board, telling it to do a vertical retrace. In part b), we see the beam returning to the top of the screen. The beam is blanked during the vertical retrace, so that we won't see the zig-zag lines on the screen. (Note that horizontal retraces continue while this is taking place).

   One complete frame includes the following;

```
      - 12 Start Scan Lines
      - 378 Active Scan Lines
      -  6 Extra Scan Lines
      - 19 Vertical Retrace Scan lines
 ----------------------------------------
       415  = scan lines per frame
```

Each scan line is divided into its active (displayable) part, and the blanked (horizontal retrace) part as follows:

```
      - 80 displayable character times
      - 35 horizontal retrace character times
     ---------------------------------------
      115  = visible and non-visible characters per scan line
```

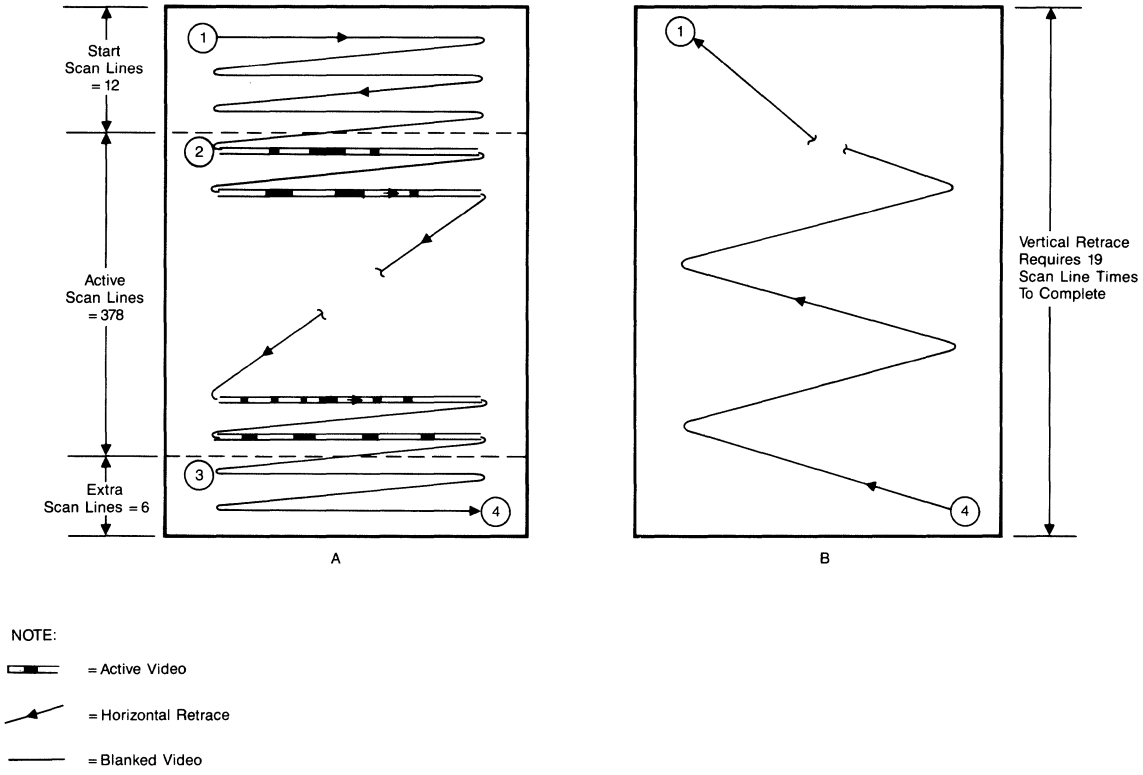


Figure 3-11. Alpha Display CRT Scanning

ALPHA SCREEN FORMAT. Observing part a) of figure 3-12 we see that the alpha screen consists of a matrix of characters 27 rows X 80 columns. From here on, we will refer to them as Character Rows and Character Columns. Following is a table describing the different areas of the screen:

Description	Row Number
24 user rows	1-24
2 soft key rows	25-26
1 status row	27
27 rows = total displayable rows in one frame	

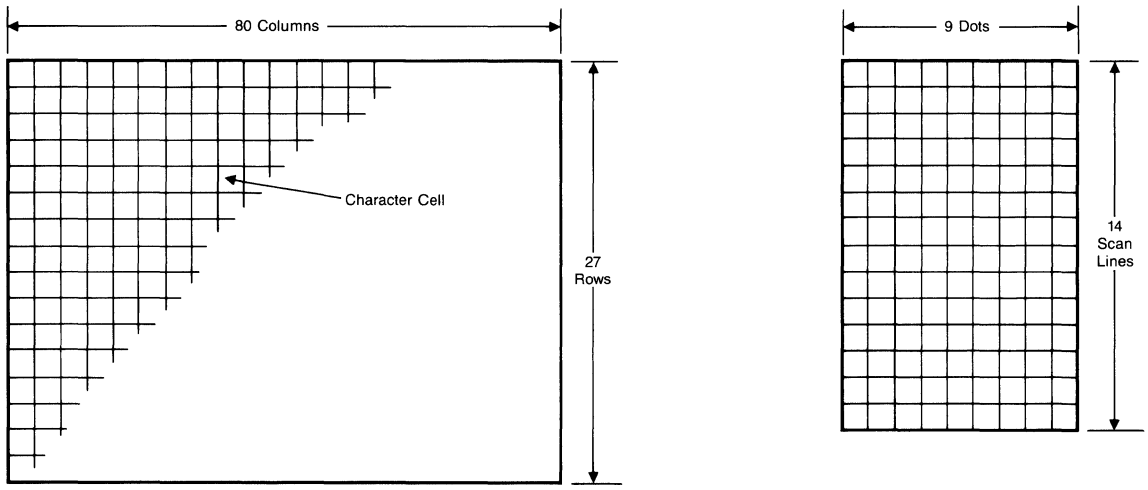


Figure 3-12. Alpha Character Matrix and Cell

DISPLAYING ALPHA CHARACTERS

- The dot pattern for every character is stored in a Character ROM (Read Only Memory).
- Characters are displayed one scan line at a time. For example, in order to display one character row (80 characters), the following must be done:

  1. Fetch the 1st scan line pattern of char #1 from the Character ROM, then send it out to be displayed.
  2. Fetch the 1st scan line of the 2nd character, and send to the display.
  3. Repeat this until scan line 1 of all the 80 characters has been displayed.

4. Now perform 1,2,3 again, but with scan lines 2 and again with 3, etc.. up to scan line 14 which completes the character row.

o To display the entire screen of 27 character rows the above steps 1-4 would be performed for each character row, a total of 27 times.

ALPHA CHARACTER CELL FORMAT. Part b) of figure 3-12 indicates the dot matrix which makes up each character, known as the character cell. There are 14 scan lines of 9 dot columns in each cell.

The HP 150 character cell has the following components:

```
Row
  1    x x x x x x x x x          Note:
  2    x o o o o o o o x          x  =  inter row/column
  3    x o o o o o o o x                spacing
  4    x o o o o o o o x          o  =  character dots
  5    x o o o o o o o x          s  =  space
  6    x o o o o o o o x          cu =  top half of double
  7    x o o o o o o o x                cursor, underline
  8    x o o o o o o o x                or underhang
  9    x o o o o o o o x          c  =  bottom half of
 10    x o o o o o o o x                double cursor or
 11    x o o o o o o o x                inter-row spacing.
 12    x s s s s s s s x
 13    c u c u c u c u c
 14    c c c c c c c c c

       1 2 3 4 5 6 7 8 9   : Columns
```

- o Basic character cell contains the main body of the character called the Character Font. (All HP 150 characters are in a 7X10 font.)
- o Inter-row spacing is accomplished by blanking the first and last columns of the character cell
- o Scan line 11 is called the Print Line, on which lower and upper case characters are positioned.
- o Inter-column spacing is ensured by not putting character information in rows 1 and 14. (Except on certain Roman Extention characters where ascenders include row 1.)
- o Underline occupies row 13
- o Descenders reach down to row 13

There are two types of cursors on the HP 150:

- The double scan line cursor occupies rows 13 and 14. It is a positive cursor, which means that it is logical OR-ed with whatever dots that may try to occupy the same row.

- The "Blob" cursor takes up the entire character cell. It is an invertible cursor, which means that it will invert whatever dot pattern that may be inside the cell.

- Both cursors are blinking types.

Refer to figure 3-13 for a character cell example. Here we see the letter "A." As stated above, it takes up only a 7X10 space, leaving room for inter-character spacing. A few of the dots do not start at the beginning of the designated dot position. These are shifted over by a half-dot position to make the slanted lines look much smoother.

Summary of Video Rates

Frame Rate	60 Frames/Second
Scan line Rate	415 Scan Lines/Frame
Character Rate	115 Characters/Scan line
Dot Rate	9 Dots/Character

Multiplying the above numbers together will give a dots/frame rate of 25,771,500 (or 25.7715 MHz).

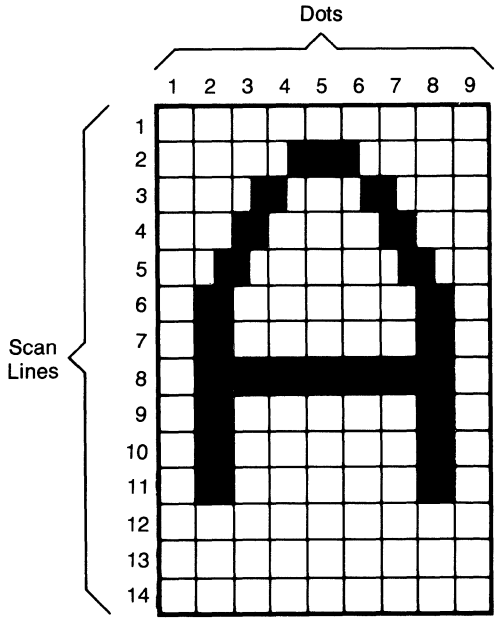


Figure 3-13. Character Cell Example

ALPHA VIDEO ENHANCEMENTS. The HP 150 supports the following types of enhancements:

- inverse
- underlined
- blinking
- half-bright
- security
  (Only alpha information is blanked out.
  It will not have any effect on other enhancements
  in the same character cell.)

Any combination of the above enhancements may be defined for a character cell.

GRAPHICS DISPLAY. Graphics display is stored on the HP 150 using a RAM array, with each bit representing a graphics dot. The 8088 microprocessor writes the dot information into RAM, while the GDC3091 (U38) Graphics Display Controller fetches this information in a continuous manner. Every fetch involves the reading of one word (2 bytes) from RAM, and once the data enters the GDC3091, it is serialized and shifted out at graphics clock rate. In this manner, all graphics scan lines are displayed.

Graphics has a resolution of 390 vertical X 512 horizontal dots. Graphics dots are 1 1/2 times as thick horizontally as alpha dots. Since the Alpha Display only has 378 scan lines, the Graphics Display area overlaps it by 6 scan lines on top and bottom of the screen. Because they are wider, the 512 graphics dots take up more room than the 720 alpha dots, and graphics overlaps alpha by about 16 grapics dots in the horizontal direction. Although the two displays are concentric in the vertical direction, they are slightly mismatched horizontally, as shown in figure 3-14.

The physical size of the display is:

Type	Horizontal (mm)	Vertical (mm)
ALPHA	150	116
GRAPHICS	160	120

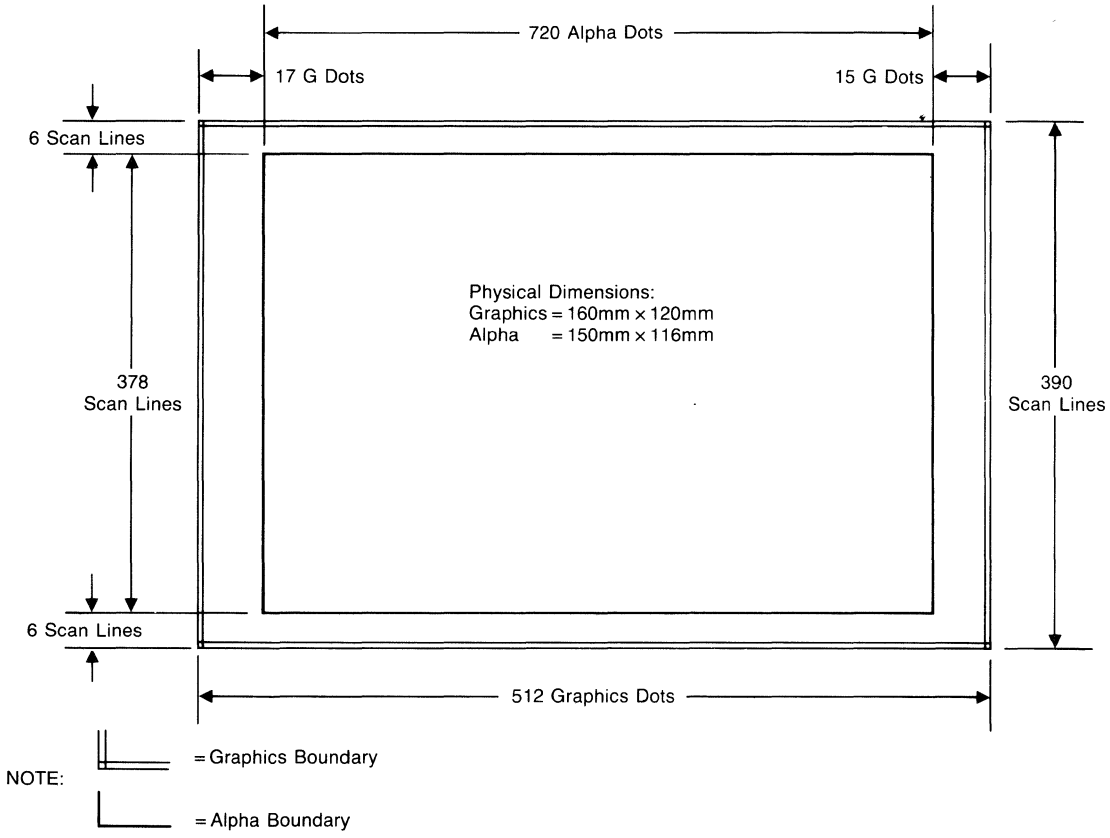


Figure 3-14. Display Specifications

Video Board Overview

The video board contains the digital logic used to display alpha and graphics information on the screen. Alpha information is addressed from a plane separate from graphics information. To do this, it performs the following functions:

1. Stores the information to be displayed in alpha and graphics RAM.

2. Produces an alpha dot stream.

3. Produces a graphics dot stream.

4. Mixes the two dot streams and sends the resulting control signals to the sweep:

   NFB not full bright
   NHB not half bright.

5. Provides deflection control signals used by the sweep.

   NVSYNC vertical sync
   HSYNC horizontal sync.

Figure 3-15 shows the major components of the video board and the flow of data and control information between them. The following list provides a brief description of each of these components.

1. Processor Request and Wait Generation:
   Decodes 8088 access to the alpha or graphics RAM producing signals used by the RAM controllers. Also inserts necessary processor wait states.

2. Graphics RAM interface:
   Interfaces the 8088 and the Graphics Display Controller Chip (GDC-3091) to the graphics RAM.

3. Graphics RAM:
   32K x 8 block of dynamic RAM which is used to store graphics information. Because the graphics is a bit map display the information is stored in that format. A portion of the RAM is also used by the alpha video firmware to store variables.

4. Graphics Display Controller Chip:
   Custom gate array graphics controller which retrieves data from the graphics RAM and forms the graphics dot stream. Also provides graphics RAM timing signals.

5. Alpha RAM controller:
   Interfaces 8088 to the alpha RAM and to the SMC 9007 video controller. Also interfaces the SMC 9007 to the alpha RAM.

6. Alpha RAM:
   12K x 8 block of static RAM which stores the alpha information to be displayed.

7. SMC 9007 Alpha Video Controller:
   VLSI video controller which retrieves data from the alpha RAM and provides display signals needed to generate the alpha dot stream.

8. Character ROM:
   16K x 8 ROM which contains template for each alpha character. The ROM contains eight character sets. Each set contains 128 characters.

9. Alpha character display hardware:
   This hardware uses information from alpha RAM and the character ROM along with signals from the SMC 9007 and generates the alpha dot character stream. This stream does not include any enhancement information.

10. Enhancement decoding logic:
    This hardware decodes the enhancement information. It produces signals used by the mixing hardware.

11. Dot stream mixing hardware:
    Mixes the alpha and graphics dot streams with the enhancement information to obtain the signals NFB and NHB which are sent to the sweep.

12. Clock generation:
    Produces various clock signals which are used by the other sections of the board.

The following discussion outlines the events which must take place to obtain the video dot stream and other signals sent to the sweep. It also highlights the key points of implementation.

To obtain an alpha dot stream the following happens:

1. The 8088 writes control information to the SMC 9007 to initialize the controller. This establishes the VSYNC and HSYNC signals sent to the sweep as well as the other control signals produced by the SMC 9007.

2. The 8088 writes the characters and enhancements to be displayed to the alpha RAM along with information needed by the SMC 9007 to retrieve the characters.

3. SMC 9007 retrieves data from the alpha RAM and latches it into discrete TTL latches.

4. This data is used to address the character ROM and generate the enhancements associated with the character.

5. The dot stream generated from the character ROM information and the enhancement information is combined to form the alpha dot stream.

NOTES: 1. Accesses to the SMC 9007 and the alpha RAM by the 8088 are synchronized to the SMC 9007 character clock. Also accesses by the SMC 9007 to the alpha RAM are synchronized to the character clock.

2. 8088 and SMC 9007 alpha RAM accesses are time multiplexed within the character clock. In other words, part of the character clock is alloted to 8088 accesses and part is alloted to the SMC 9007.

To obtain the graphics display:

1. The 8088 writes information to be displayed into graphics RAM.

2. The Graphics Display Controller Chip retrieves the information and forms the graphics dot stream.

NOTES: 1. The signal which initializes the Graphics Display Controller Chip is derived from signals generated by the SMC 9007. This means that the SMC 9007 must be initialized before the graphics display is enabled or the graphics RAM is accessed.

2. Accesses to the graphics RAM by the 8088 and the Graphics display controller are syncronized to the graphics dot clock. The accesses are also time multiplexed within each graphics cycle, where one graphics cycle is 16 graphics dot clocks in length.

To obtain the sweep signals:

1. The alpha and graphics dot streams are combined using combinatorial logic. This logic produces the NFB and NHB signals which are sent to the sweep.

2. The NHSYNC and NVSYNC signals are produced by the SMC 9007 and are buffered before being sent to the sweep.

Notes on clock generation:

1. The 8 MHz system clock is used by the request and wait generation circuitry to aid in sychronizig the 8088 RAM accesses to the video clocks.

2. All clocks on the video board are derived from a 25.7715 MHz crystal.

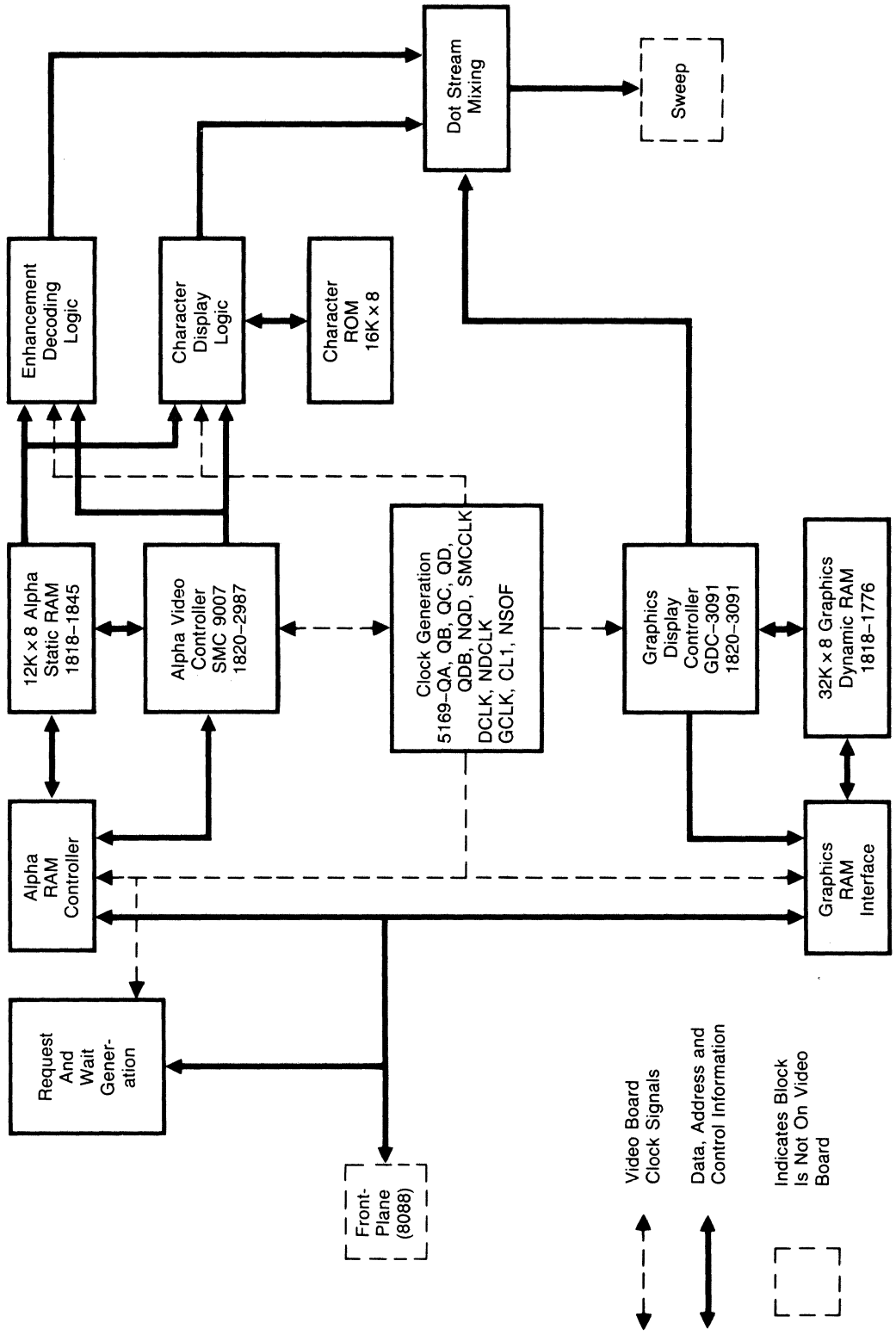


Figure 3-15. Video Board Overview

KEYBOARD AND TOUCHSCREEN SUBSYSTEM

Keyboard

ELECTRICAL INTERFACE. The keyboard interfaces with the 8088 via an 8041 microcomputer that creates the timing signals needed for communications. Power for the keyboard is delivered by the processor board through the keyboard cable. The 8041 delivers a clock and sync signal to the keyboard while monitoring a key acknowledge return signal from the keyboard on one of the 8041's timer inputs.

The following signals are connected to the connector jack on the keyboard:

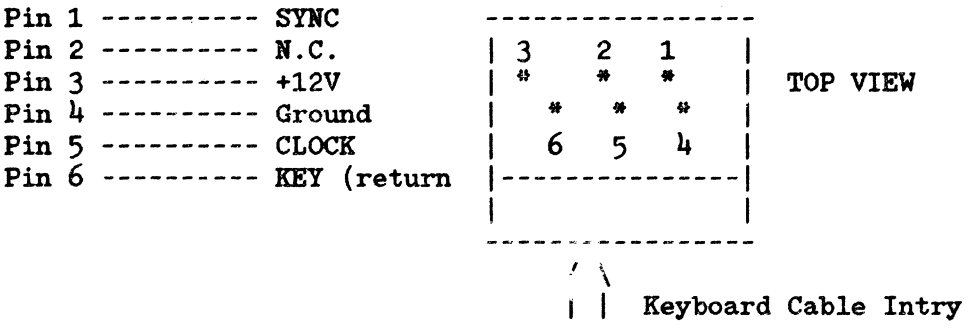


Figure 3-16. Keyboard Jack Detail

The 8041 creats a CLOCK signal that is non-periodic that averages just under 5 KHz. The shortest time period is around 190 microseconds with a duty cycle near 50%. The SYNC signal (active high) is designed to reset the keyboard to a known state with respect to the 8041. The KEY line is a return signal from the keyboard telling the 8041 a key was depressed.

KEYBOARD OPERATION. The 8041 clocks the keyboard continually by pulsing the CLOCK line and sends out a synce pulse on the SYNC line after all the keys have been scanned. The sync pulse keeps the 8041 in sync with the keyboard.

After the 8041 delivers a sync pulse to a seven bit counter on the keyboard, it resets it to zero. On the very next negative edge of the clock this counter begins to count. The three least significant bits of the counter control the inputs to an 8 to 1 multiplexer which monitors the 8 rows of the keyboard matrix (each clock pulse causes the multiplexer to scan a different row). The four most significant bits are controlling two BCD to decimal decoders which strobe each column of the matrix. The inputs to the 8 to 1 multiplexer are pulled down via a resistor. For a particular column address, an output of the decoders will be high. If a key in that particular column is depressed, one of the eight inputs to the 8 to 1 multiplexer will go high. As the least significant bits of the counter count through all eight rows of that column, the output of the multiplexer will go high when the row of depressed key is scanned. This logic one signal is returned to the 8041 via a NAND buffer. The 8041 keeps track of how many clocks were sent to the keyboard and can determine which key was depressed when the logic one was sent on the KEY line since there is a

one-to-one correspondence to the number of clock pulses sent and each key switch.

Touchscreen

NOTE

The information presented here may be covered by one or more Hewlett-Packard patents.

The touchscreen allows the determination of the X-Y coordinate of an object touching the screen. This is accomplished by placing infrared emitters along two sides of the grid, and photo-detectors along the other two sides, opposite the emitters. By turning on each emitter and checking the output of the opposite detector, it can be determined which pairs are blocked. A complete scan of the pairs results in an image of the object, allowing the determination of an X-Y coordinate.

MECHANICAL DESCRIPTION. The touchscreen consists of a printed circuit board with a center cut-out. This mounts slightly in front of the CRT face, inside the bezel. The infrared beams pass through holes in the bezel and across the CRT face.

Connecting the touchscreen p.c. board with the terminal is one 10 conductor cable. It plugs into the touchscreen board at the upper right hand corner, and into the frontplane next to the sweep board connector.

For ESD protection, the inside of the bezel insert is coated with a conductive paint. Contacting this paint is a spring clip mounted on the back of the upper right corner of the board. This clip is bolted to the PCA and connects through the pad to a quick disconnect on the front of the board. A ground cable runs from the quick disconnect to the chassis.

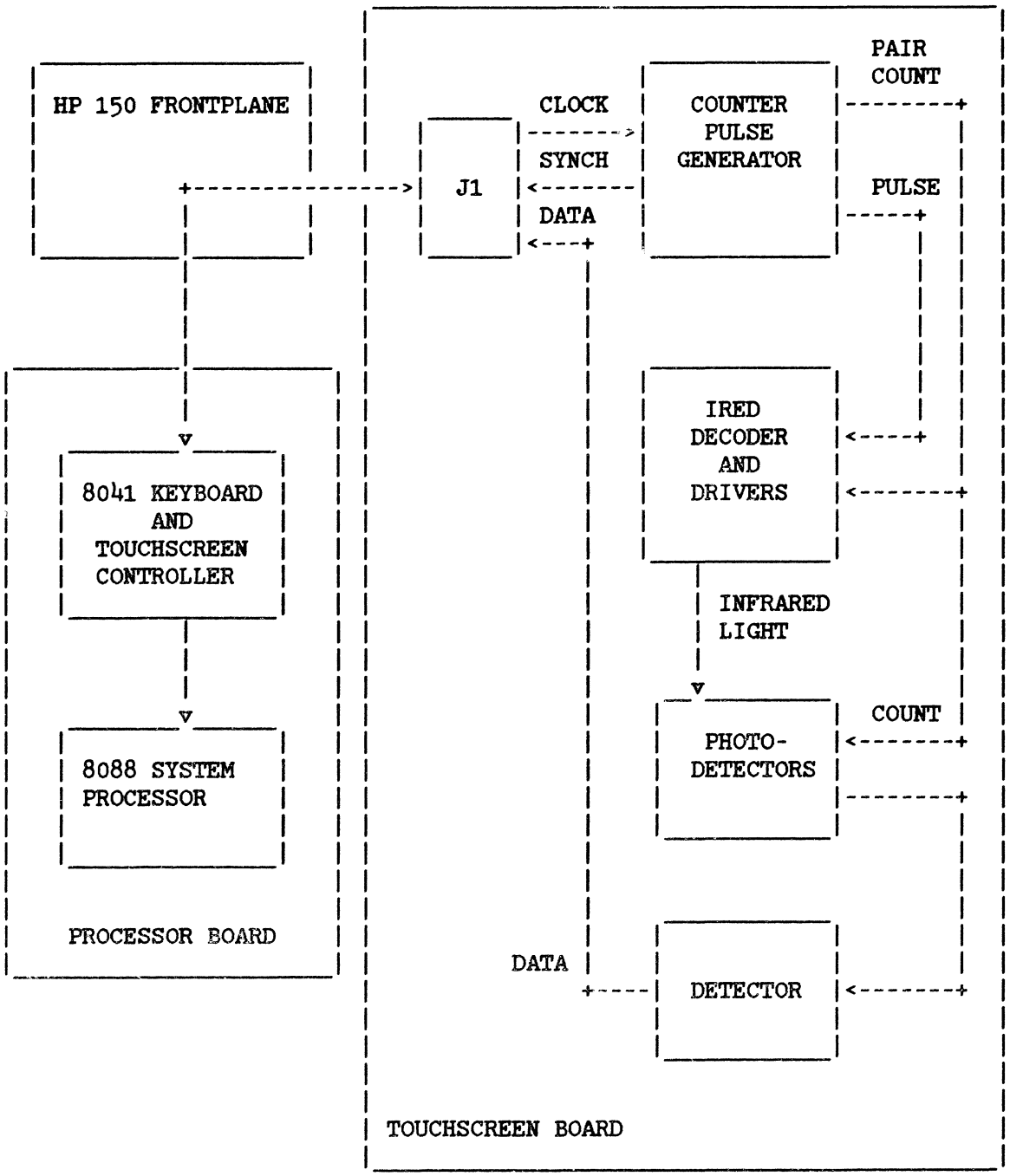


Figure 3-17. Touchscreen Block Diagram

SPECIFICATIONS

Resolution: 27 rows by 40 columns

Power Consumption: +12V @ 70ma

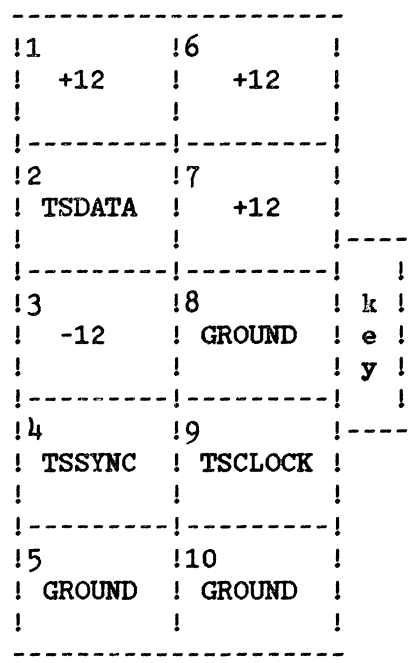
Interface: 6 signals;

1. +12
2. ground
3. -12
4. TSCLOCK: should be driven with an open collector buffer pulled up on the T.S. board to 10 volts. (Period of 1.4 milliseconds or greater. active low, low time to be > 1.5 microseconds and less than 10 microseconds.)

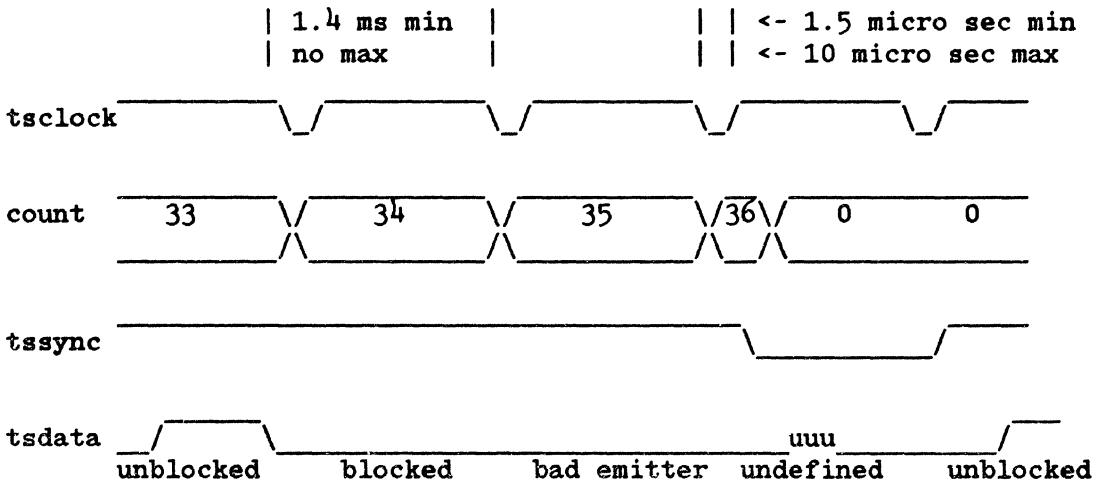
5. TSDATA: high output is 10 volts. (Data out, low indicates that the currently addressed pair is interrupted. High indicates that the pair is unblocked.)

6. TSSYNC: high output is 10 volts. (Synch out, active low, indicates that the ts counter is currently on 37 (43 base 8) and will roll over to zero on the next clock.)

Connector: J1 10 pin connector with key.



TIMING



- Delay from falling edge clock to data valid = 0.8 msecs maximum.
- No pair exists at decimal count 21, thus data will always be high at this count.

INTERFACE DESCRIPTION. In the HP 150, a single chip microcomputer, an 8041, provides the interface to the touchscreen. Within the 8041, a software count is synchronous with the touchscreen hardware counter. As the 8041 sends clocks to the touchscreen, it increments this software count and checks the data line. When the data shows a blocked pair, the software count specifies which pair is blocked. By keeping track of which pairs were interrupted in a scan cycle, the 8041 can determine if a valid hit has occurred. If so, the hit is reported as one of 27 rows and one of 41 columns. (0-26, 0-40).

The general scheme for this is as follows:

1. While sync not active send clock (*synchronize count*)
2. Set count = 00
3. For count = 0 to 36
   send clock
   wait 1.4 milliseconds
   check data
   if data = block store count
   end
4. Check synch
5. If synch is not active then error - GOTO step 1
   else check blocked count numbers for a valid hit
6. If valid hit then report to system processor
7. GOTO step 2.

Valid Hits: At least one interrupted pair in the column range and at least one interrupted pair in the row range. More than one pair, in row or column space, is okay as long as the multiple, interrupted pairs are adjacent. (Example: pair 3 interrupted, and pair 5 interrupted is an error condition.)

No Hit: This would be a scan cycle which finds no pair interrupted.

Error Conditions:

1. An interrupted row pair, but no interrupted column pair.

2. An interrupted column pair, but no interrupted row pair.

3. Multiple hits: If any two interrupted row pairs are separated by one or more uninterrupted pairs. Likewise for columns (i.e., two distinct objects are touching the screen).

4. Out of synch: At the end of a scan cycle, the synch should be active. If this does not occur, a reset should be done.

   If an error condition is found, no hit is reported.

RESOLUTION VERSUS NUMBER OF PAIRS. The resolution of the touchscreen is greater than the number of pairs used. This is accomplished by averaging the data. That is, if two adjacent pairs are interrupted, the row or column between the two is the one desired. Thus the resolution is effectively doubled, except for edge effects.

This requires that the distance between the adjacent pairs be less than the minimum stylus diameter.

Thus, if any single row or column is interrupted, that is the one desired. If an even number of adjacent beams are interrupted, the result is the center, 'half', address. If an odd number of pairs are interrupted, the result is the center beam.

PAIR ADDRESS TO ROW, COLUMN NO. CONVERSION. The pairs are positioned above and below the CRT as follows:

```
---          1         2         3         4         5
3210123456789012345678901234567890123456789012345678901234567 89
 x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x
 0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15

                    Column No.     6         7         8
                                   012345678901234567890
                                     x   x   x   x   x
                    Pair No.        16  17  18  19  20
```

Thus, across the top, the pairs are positioned every fourth column. This leads to a resolution of 40 vertical divisions, each division = 2 columns.

The pairs are actually offset to the space between 2 adjacent columns, starting at the space between columns -2, -1 and repeating every fourth column until the space between column 78, 79. This causes an offset which should be corrected at the system processor level. The effect is that the 8041 will report column pairs in the range 0 to 40. This is one more division (two columns) than on the display. To correct for this, the following should be done:

```
Multiply the column report by 2 => changes range from 0-40 ->  0-80
  Subtract 2                    => changes range from 0-80 -> -2-78
  If negative then set result=0 => changes range from -2-78 -> 0-78
```

This corrects for the offset of the touchscreen hardware with respect to the display.

The pairs are positioned along the sides of the CRT as follows:

T.S. Pair	Display Row
0	0
	1
1	2
	3
2	4
	5
3	6
	7
4	8
	9
5	10
	11
6	12
	13
7	14
	15
8	16
	17
9	18
	19
10	20
	21
11	22
	23
12	24
	25
13	26

The rows are a straight forward mapping of the row number reported by the 8041 into the rows of the display. The report need only be multiplied by 2.

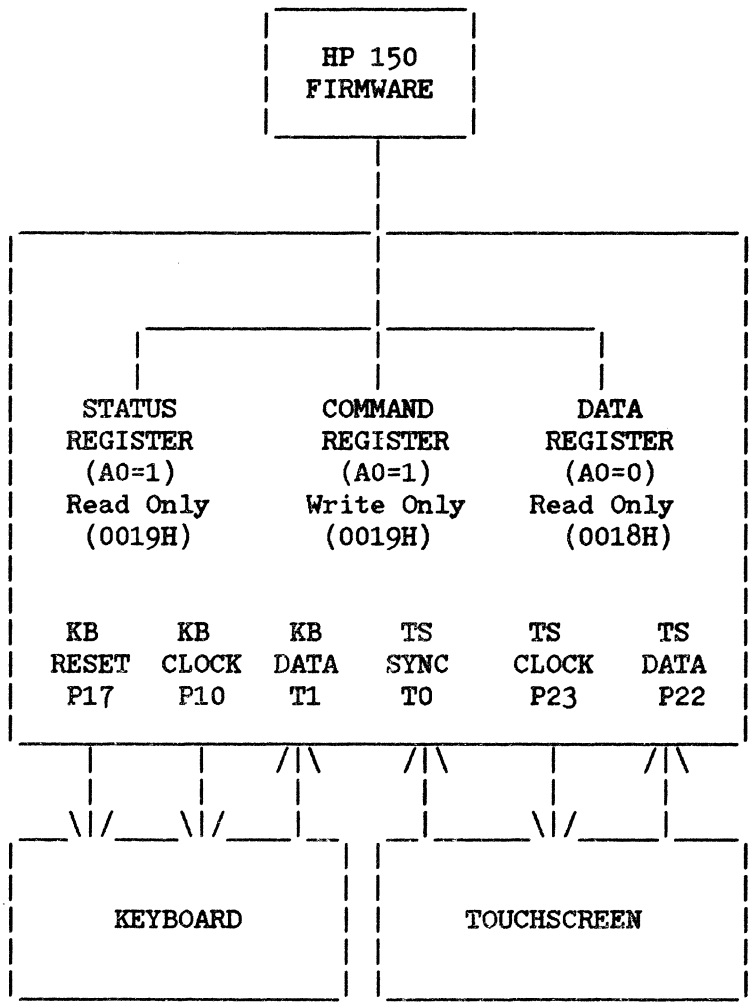
8041 Keyboard/Touchscreen Scanner

A description of the operation of the 8041 peripheral processor used to scan the keyboard and touchscreen follows. A complete description of the commands used to control the 8041 is included.

NOTE

The information presented in this manual with respect to the touchscreen may be covered by one or more Hewlett-Packard patents.

BLOCK DIAGRAM. The keyboard and touchscreen are both scanned by an 8041 peripheral processor.



STATUS REGISTER (I/O PORT 0019H). The 8041 status register contents are obtained by reading from the 8041 at I/O port address 0019H. It has the following format:

Bit	Interpretation
7 - 4	Four status bits controlled by the 8041 program. These are used to qualify the data values sent to the HP 150 processor by the 8041.
3	F1 flag. Used internally by the 8041.
2	F0 flag. Used internally by the 8041.
1	IBF flag. Set when the HP 150 processor writes to the 8041 and cleared when the 8041 accepts the data.
0	OBF flag. Set when the 8041 has data available for the HP 150 processor and cleared when the host reads from the 8041 with A0 = 0.

INITIALIZATION. The following actions will leave the 8041 in its initialized state:

- o Power on
- o Hard reset from keyboard
- o Hard reset command from HP 150 processor
- o Initialization command from HP 150 processor

After initialization, the 8041 will not scan the keyboard or touchscreen until an "enable scanning" command is given. About 200us after initialization commences, status bits 7-4 will be 0001 if the reset was from power-on or 0010 otherwise. The initialization continues and attempts to synchronize with the touchscreen. The entire initialization takes up to 100 ms.

The recommended start-up procedure for the 8041 after any of the four initialization actions listed above is:

1. Wait 100 milliseconds for the 8041 to initialize itself and check the touchscreen.

2. Send self-test command and wait for results to come back.

3. Send indentify keyboard command and wait for results to come back.

4. Enable scanning.

8041 COMMANDS (I/O PORT 0019H). These commands must only be written to the command buffer of the 8041 (I/O port 0019H).

Command Code	Function
08H	Stop bell immediately. Next bell will function normally.
10H	Release interrupt line. 8041 will negate its interrupt request to the host.
20H	Hard reset. 8041 will assert its reset output to the HP 150 processor and initialize itself to power-on state with the exception of 8041 status flags (see preceding discussion on 8041 status register). Scanning must be re-enabled for keyboard or touchscreen input.
21H	Identify keyboard. 8041 will respond with either 7FH or FFH depending on whether the keyboard identification diode is present or not. The 8041 generates an interrupt to the host when the ID is ready. Status bits 7-4 will be 0000 if touchscreen is defective and non-zero if the touchscreen is working.
22H	Keyclick off. 8041 will not make keyclick sound when keys are pressed.
23H	Keyclick on. 8041 will make keyclick sound.
24H	Enable Scanning. After power-on or hard reset, 8041 will not scan keyboard or touchscreen until command is given by the host.
25H	Initialize 8041 to power-on state except for status flags. Scanning must be re-enabled by HP 150 processor for keyboard or touchscreen input. This command does not reset the HP 150 processor.
26H	Self-test 8041. The 8041 will test its RAM and ROM and respond with F9H if it passes or F8H if it fails. No HP 150 processor interrupt will be generated; the status register must be polled until data is available.
28H	Stop key repeat. Any key on the keyboard will auto repeat after 500 ms if held down. If the HP 150 processor does not want the current key to auto repeat, it must send this command to stop it.
2AH	Disable hard reset. This command will disable the hard reset that is initiated by the control/ shift/reset combination.
2BH	Enable hard reset. This command enables the keyboard hard reset. (Default)

30H - 3FH — Beep bell. The bell duration is about 100 ms and its period is determined by the lower four bits of the command code. With a 5 MHz 8041 clock, the frequency is about 290 Hz for 30H and about 1.7 KHz for 3FH.

40H — Do one keyclick immediately. Ignored if keyclick is disabled.

60H — Touchscreen detector pairs report. This command is used to determine if any LED/Transistor pairs appeared to be blocked or bad during initialization. Two data bytes are always returned after this command is given. If either or both are not 0FFH, then their values are the addresses of the blocked pairs. If both are 0FFH, then there are no blocked pairs. No interrupt is generated for these two bytes; the 8041 must be polled for them. This command must be given after initialization and before scanning is enabled because it will interfere with the interrupt system if the 8041 is attempting to report keycodes.

Table 3-1. Ired vs. PT

Decimal Count	Address Hex	Address Octal	Ired CR_	Photo-Transistor Q-
0	00	00	36	21
1	01	01	35	20
2	02	02	34	19
3	03	03	33	18
4	04	04	32	17
5	05	05	31	16
6	06	06	30	15
7	07	07	29	14
8	08	10	28	13
9	09	11	27	12
10	0A	12	26	11
11	0B	13	25	10
12	0C	14	24	9
13	0D	15	23	8
14	0E	16	22	7
15	0F	17	21	6
16	10	20	20	5
17	11	21	19	4
18	12	22	18	3
19	13	23	17	2
20	14	24	16	1
21	15	25	blank	blank
22	16	26	2	22
23	17	27	3	23
24	18	30	4	24
25	19	31	5	25
26	1A	32	6	26
27	1B	33	7	27
28	1C	34	8	28
29	1D	35	9	29
30	1E	36	10	30
31	1F	37	11	31
32	20	40	12	32
33	21	41	13	33
34	22	42	14	34
35	23	43	15	35
36	24	44	reset	reset

KEYBOARD AND TOUCHSCREEN DATA INPUT (I/O PORT 0018H). When data is available as a result of key presses or touchscreen touches, the 8041 will write to its output register and signal an interrupt request to the HP 150 processor. Data is buffered one byte deep in the 8041. If the previous byte written by the 8041 to its output register has not been read by the HP 150 processor, it will wait. Otherwise, it will resume scanning. Bits 7-4 of the status register are valid when the data register is valid. They are used to indicate the type of data available. The status values are:

0000 - Key address. If bit 7 of the data byte is 0, the key was depressed, else the key was released.

Exception: If the data is in response to an Identify Keyboard command, then 0000 means the touchscreen is not connected.

0001 - Power on (valid until first write).

0010 - Reset (valid until first write).

0011 - Touchscreen release code. Data is 0.

0100 - Touchscreen row address. Same as screen row.

0101 - Touchscreen column address. Same as screen column/2.

1000 - Calculator command is complete. Data is 0 if command was not read from calculator.

1111 - Only occurs after Identify Keyboard command; indicates that touchscreen is connected.

NOTE

These status bits are valid only when data is available (OBF flag = 1). The only exception to this is the power-on/reset status which become valid 200 microseconds after reset or power-on and remains valid until the first write to the 8041.

After reading the 8041 data register, the host MUST send a release interrupt command to the 8041 to acknowledge receipt of the data. (Except for self-test results and touchscreen bad pair addresses, which are reported without setting the interrupt lines.)

Data from the touchscreen always comes in pairs of bytes, each row address will be followed by a column address. As soon as the touchscreen scanner completes a scan without any touches, the 8041 will report a touchscreen release code. Touchscreen row/column reports will be generated each time a new touched position is detected. The only way that the same position can be reported twice in a row, is when the screen is released between touches. In this case, a release code will be sent between touch reports.

DATACOMM SUBSYSTEM

General Description

The data communication (datacomm) electronics of the HP 100 Series Personal Computer allows the computer to communicate with other computers, mainframes, or peripherals via a serial data stream. The HP 150 has two serial EIA Standard RS232C (CCITT V.22) datacomm ports; Port 1 which is designed for communication with other computers either directly or using a modem, and Port 2 which is designed primarily for use with peripherals such as RS232C printers, but also can be used for computer communications.

The principal components in the datacomm circuit are the baud rate generator chip and the serial controller chip, both of which are on the I/O bus of the 8088. The baud rate generator chip supplies the timing needed for data serialization for both ports independently and the Multiprotocol Serial Controller chip (MPSC) controls both of the datacomm ports. During communications, the 8088 writes and reads 8-bit data to and from the MPSC which serializes or deserializes the data from the proper port. The rate of data serialization is governed by the timing sent to the MPSC by the baud rate generator.

RS232C/422 Datacomm Module Connector

A 36 pin connector, J3 on the Processor PCA, interfaces the RS232C/422 datacomm module to the system. The connector signals are:

1.	+12V	13.	N.C.	25.	DATA 7
2.	+5V	14.	NBRD	26.	DATA 6
3.	NRxC(1)	15.	GND	27.	DATA 5
4.	NRR(1)	16.	---	28.	DATA 3
5.	GND	17.	---	29.	DATA 1
6.	RD(1)	18.	---	30.	x16 CLK(1)
7.	NTR(1)	19.	-12V	31.	NSELB
8.	GND	20.	+5V	32.	NWRT
9.	DATA 4	21.	NTxC(1)	33.	NRST
10.	DATA 2	22.	NCS(1)	34.	---
11.	DATA 0	23.	NRS(1)	35.	---
12.	GND	24.	SD(1)	36.	---

The numbering scheme for the connector goes from 1 through 36 with pins 16, 17, 18, 34, 35, and 36 undefined and physically absent on the 36 pin connector. The signals are defined as:

DATA 0 - The eight bit interconnect to the 8088 I/O bus.
DATA 7

NRxC(1) Receive timing input to the 7201 datacomm controller
NTxC(1) Transmit timing input to the 7201 controller
x16 CLK(1) Baud rate generator output

NRR(1) Receiver Ready
NTR(1) Terminal Ready
NCS(1) Clear to Send
SD(1) Send Data
RD(1) Receive Data
NRS(1) Ready to Send

NRST System Reset
NBRD 8088 read cycle
NWRT 8088 write cycle
NSELB (not used)

Baud Rate Generator

The baud rate generator used in the HP 150 is the SMC 8116T Dual Baud Rate Generator which can supply two independent baud rates based on a single input clock. It consists essentially of a pair of programmable dividers using an input frequency of 4 MHz. The user defines a baud rate which the 8088 translates into an 8-bit value which is written into the 8116T at I/O address XX0C hex, the most significant 4 bits of which set the baud rate for Port 2 and the least significant 4 bits set the baud rate for Port 1. The timing outputs then connect to the proper port or channel timing inputs on the MPSC.

The 8116T, in this application, generates clock frequencies for asynchronous data communications only. Asynchronous communication requires that the input clock frequency be 16 times the transmission baud rate, e.g. for a baud rate of 300 baud, a clock frequency of 4.8 KHz is required to serialize and deserialize the data. See the "Memory and I/O Mapping" section of this manual for 8116T programming information.

The clock frequencies for Port 2 tie directly into the Transmit and Receive clock inputs of the MPSC and are always the baud rate times 16.

The clock frequencies for Port 1 go to the datacomm interface for Port 1 and into a multiplexer, the outputs of which tie to the Transmit and Receive clock inputs of the MPSC. The multiplexer allows the selection of the 8116T clock, the times-one clock, or the Send timing clock as the Transmit clock input to the MPSC. It also selects the 8116T clock, the times-one clock, or the Receive timing clock, respectively as the Receive clock input to the MPSC. A latch drives the mulitplexer select inputs and is accessed by writing the bit patterns as described in the "Memory and I/O Mapping" section of this manual.

The times-one clock is derived by dividing the 8116T clock output by 16 giving a clock of the same frequency as the transmission baud rate. This clock is available to support some modems which require a times-one clock for data transmission. Even though the Transmit and Receive clocks may be configured to use the times-one clock, the communication protocol always assumes asynchronous communication.

Multi-Protocol Controller

All of the data communications in the HP 150 are controlled by the MPSC, which is a dual-port communications chip. Channel A on the chip controls communication to Port 1 while channel B controls Port 2. The chip is capable of being configured to do all of the data serialization and deserialization, stop/start bit insertion and deletion and parity bit insertion and deletion. The chip also controls standard communications status lines on each port through internal registers such as Terminal Ready (TR) (CD), Clear to Send (CS)(CB), Receiver Ready (RR)(CB), and Request to Send (RS)(CA). The data is transmitted from each port on the Send Data line (SD)(BA) and received on the Receive Data line (RD)(BB). The data received or transmitted is accessed by the system CPU by addressing the data register for the proper port. The chip also provides a FIFO three-byte receive buffer to prevent incoming data overrun.

The chip is also capable of interrupting the CPU upon receipt or transmission of a character. The Interrupt Request output is tied to the second priority interrupt input of the interrupt controller. This is an open-collector output and is shared with the open-collector datacomm request line coming from the accessory slots (NDCOCINT). In addition to interrupts on character transmissions, the MPSC also generates interrupts on parity errors and changes in status of the communication control line inputs, freeing the CPU from polling these on each character transmission.

The internal logic of the MPSC chip (aside from transmit and receive logic) runs on the system 4 MHz clock (PCLK) and the chip is reset by system resets (NRST). The Transmit and Receive clock signal inputs used for channel B (Port 2) are tied together and are always derived from the baud rate generator times-16 clock. The Transmit and Receive clocks for channel A are driven independently from the Port 2 interface circuit (see Baud Rate Generator). The chip's Ready outputs for synchronization with the CPU are not used since this function is accomplished by always asserting an extra cycle in I/O accesses which should satisfy access times for this part. The MPSC also has Interrupt Priority inputs and outputs which are not used in this system.

Communications Interface Circuitry

The circuitry interfacing the MPSC to the communications circuitry consists primarily of line drivers (MC1488) and receivers (MC1489) which convert the MPSC TTL levels to RS232C transmission levels (+12V/-12V). The RS232C standard requires that driver outputs must not exceed a slew rate of 30 volts per microsecond. All driver outputs are tied to a 470 pf capacitor which provides an effective slew rate of 21.3 volts per microsecond maximum (assuming short circuit current). The receiver gates (MC1489) have "response control" inputs which allow filtering of high frequency noise pulses. The 470 pf capacitor used should filter noise faster than 100 ns in the 12V range. Both Port 1 and Port 2 use these parts for interfacing to the communications channel. The signal interconnection for both ports is shown in table 3-2.

Table 3-2. Datacomm Interconnect List

Connector pin	Port 1 Signals	Port 2 Signals
1	Shield(AA)	Shield(AA)
2	SD(BA)	SD(BA)
3	RD(BB)/RD.A*	RD(BB)
4	RS(CA)	RS(CA)
5	CS(CB)	CS(CB)
6	DM(CC)	DM(CC)
7	SG(AB)	SG(AB)
8	RR(CF)	RR(CF)
9	SD.A*	
10	SD.B*	
12	OCR2(SCF)	OCR2(SCF)
15	ST(DB)	
17	RT(DD)	
18	RD.B*	
19	OCD2(SCA)	OCD2(SCA)
20	TR(CD)	TR(CD)
22	OCR1(CE)	OCR1(CE)
23	OCD1(CH)	OCD1(CH)
24	TT(DA)	

* Represents RS422 signal.

The control lines OCD1, OCD2, OCR1, OCR2 and DM are not connected directly to the MPSC, but are tied to the CPU data bus through handshake capability with certain kinds of equipment that require them (e.g. printers, modems, etc.) and are all available on both ports. I/O ports 14H and 16H access these control lines on Ports 1 and 2 respectively as described in the "Memory and I/O Mapping" section of this manual.

Port 1 differs from Port 2 in two major respects; Port 1 contains timing signals on the RS232C interface as well as RS422 signals on pins that are not used by RS232C in most HP applications (those pins are reserved for Data Set Testing in the RS232C definition). The timing signals added are Transmit Timing (TT)(DA), Send Timing (ST)(DB), and Receive Timing (RT)(DD). The Send and Receive timing signals allow an external device to drive the MPSC clock inputs for serialization. The Transmit Timing signal is always the same as the MPSC input tranmission clock.

The RS422 interface consists essentially of a differential driver and receiver (75179) in place of the RS232C drivers. These drivers transmit and receive differential voltages with respect to the opposite signal line as opposed to referencing ground. Note that the RS232C signal RD(BB) and the RS422 signal RD.A are shared in this application. This is accomplished by pulling down the opposite driver input to create the proper translation to TTL level signals when the RS232C cable is connected. The selection between RS232C and RS422 is simply a matter of connecting the cable which connects to the proper signals. Since the receive signal comes into the same gate and the transmit signal drives both interfaces in parallel, the difference is transparent to the MPSC.

MEZZANINE MEMORY SUBSYSTEM

Mezzanine Memory PCA

PCA Overview

The HP 150 mezzanine memory PCA (assembly 45611-60006) houses ROM, dynamic RAM, CMOS RAM, and system status LEDs on a PCA in a mezzanine position between the processor and video PCAs. The PCA is supported by four standoffs mounted on the processor PCA and electrically interfaces to the processor PCA through a 60 pin connector. The block diagram of the PCA is shown in figure 3-18.

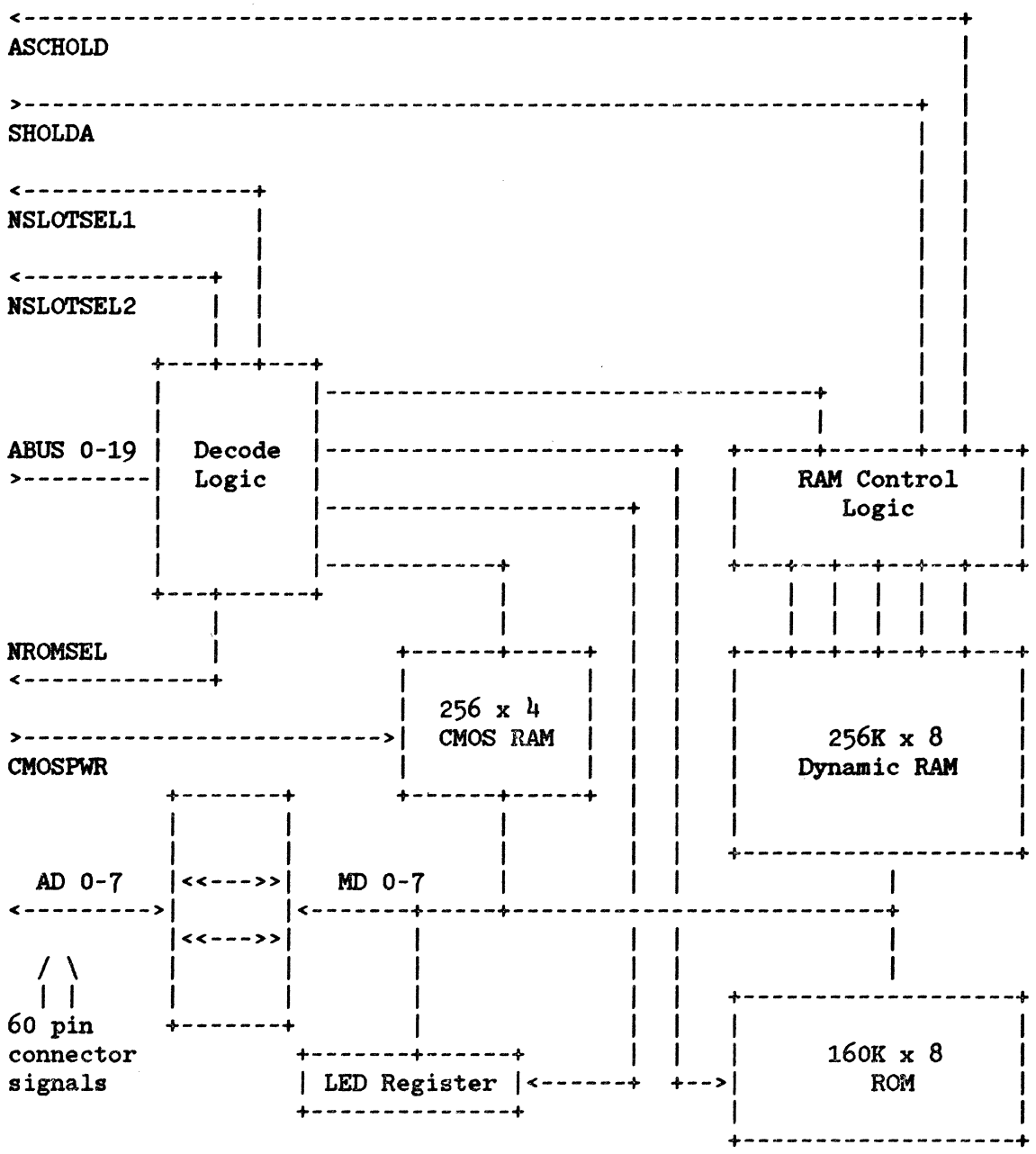


Figure 3-18. Mezzanine Memory PCA Block Diagram

Connector Signals

A 60 pin connector, J2 on the Processor PCA, connects the ROM and user RAM to the rest of the system. The signals on the interface are listed below:

Mezzanine Memory Connector Signals

60.	NSLOTSEL1	30.	+5V
59.	ABUS 11	29.	ABUS 19
58.	NSLOTSEL2	28.	ABUS 9
57.	ASCHOLD	27.	NPWRDN
56.	AD1	26.	AD0
55.	AD3	25.	AD2
54.	AD5	24.	AD4
53.	AD6	23.	DT/-R
52.	ABUS 5	22.	AD7
51.	ROMSEL	21.	ABUS 16
50.	ABUS 17	20.	ABUS 14
49.	ABUS 18	19.	BCLK
48.	ABUS 12	18.	ABUS 13
47.	NCMOSSL	17.	NCMSWRT
46.	ABUS 4	16.	NXR1
45.	+5V	15.	GO
44.	FULLMEM	14.	+5V
43.	+5V	13.	BDEN
42.	ABUS 7	12.	CMOSPWR
41.	GND	11.	GND
40.	+5V	10.	GND
39.	GND	9.	NRWRT
38.	NCMSOE	8.	SHOLDA
37.	ABUS 5	7.	ABUS 6
36.	BIO/-M	6.	NBRD
35.	ABUS 8	5.	ABUS 10
34.	+5V	4.	ABUS 3
33.	ABUS 1	3.	GND
32.	NRST	2.	NMSEL
31.	ABUS2	1.	ABUS0

The mezzanine memory PCA connector signals are described below:

ABUS 0-19 — The demultiplexed 20 bit address generated by the 8088 during a bus cycle.

AD 0-7 — Least significant 8 bits of the multiplexed data/address bus. Data is transferred to and from the memory PCA over these lines.

NSLOTSEL1	Signal indicating an address within the 64K block allocated to accessory module 1 (90000H - 9FFFFH) has been generated by the 8088.
NSLOTSEL2	Same as NSLOTSEL1 but is asserted when the address is in the address space allocated to accessory module 2 (A0000H - AFFFFH).
GO	Signal which qualifies the address generated by the 8088. GO goes high at the beginning of T2 and goes back low at the beginning of T4 of a bus cycle.
BCLK	Buffered 8 Mhz system clock.
BIO/-M	Buffered IO/-M signal from the 8088.
DT/-R	DT/-R from the 8088.
BDEN	Inverted and buffered -DEN signal from the 8088.
NBRD	Buffered -RD signal from the 8088.
NRWRT	Buffered -WRT signal from the 8088.
ASCHOLD	Asynchronous HOLD request generated by the memory PCA and sent to the 8088.
SHOLDA	Synchronized (with the 8 Mhz clock) HOLD acknowledge from the 8088.
NMSEL	Signal if high prevents write to dynamic RAM.
NXR1	Transceiver control signal generated by memory PCA.
ROMSEL	ROM access indicator which causes the processor PCA to eliminate wait state on the bus cycle.
FULLMEM	When high indicates 256K RAM loaded on the memory PCA. When low indicates 128K RAM loaded. (A pull-up resistor external to the memory PCA is used to pull high.)
NMEMRST	System reset line for the memory PCA.
NCMOSSL	CMOS RAM select line active low.
NCMSOE	CMOS RAM output enable active low.
NCMSWRT	When low causes a write into CMOS RAM.
CMOSPWR	Battery backed-up power source for the CMOS RAM.

NPWRDN	Goes low to indicate an imminent power interruption to the system and puts the CMOS RAM into the standby mode.

ROM

ROM DECODING. Five 32K x 8 ROMs contain the system firmware used by the PC. These ROMs are located in the memory map as follows:

ROM	Memory Address
(ROM 5)	0F8000H - 0FFFFFH
(ROM 4)	0F0000H - 0F7FFFH
(ROM 3)	0E8000H - 0EFFFFH
(ROM 2)	0E0000H - 0E7FFFH
(ROM 1)	0B0000H - 0B7FFFH

For a particular ROM to be selected, an address within the memory space allocated to it must be generated by the 8088, the 8088 control signal IO/-M (becomes BIO/-M after buffering for the memory PCA) must be low, and NBRD (buffered read strobe control signal fromn the 8088) must be asserted (active low). The decode circuitry for ROMs 2,3,4, and 5 decode address lines ABUS 19, 18 18, 17, 16, and 15 when BIO/-M is low and assert the chip select input of the ROM whose contents is at the address generated by the processor. The 8088 receives data from the memory (or writes to RAM or the LED register) through a transceiver on the memory PCA. This transceiver is enabled when ROM, CMOS RAM, dynamic RAM, or the LEDs are accessed.

WAIT STATE DISABLE. Every bus cycle (memory or I/O operation) of an 8088 consists of four system clocks unless logic external to the 8088 dictates that the cycle be extended by some integer number of additional clocks. The wait state generation circuitry on the processor PCA inserts a minimum of 1 additional clock period into all bus cycles unless a ROM access is being made.

ROM TIMING. The memory address from the 8088 becomes available to the memory PCA 85 nsec after the beginning of T1. The propagation delays until data is available to the processor is as follows:

ROM 1 Access		ROMs 2-5 Access	
address valid	85 nsec	address valid	85 nsec
chip select delay	31 nsec	chip select delay	29 nsec
ROM access time	200 nsec	ROM access time	200 nsec
buffer delay	12 nsec	buffer delay	12 nsec
connector delay	2 nsec	connector delay	2 nsec
total delay	330 nsec		328 nsec

The cycle is 375 nsec long (T1 + T2 + T3). The data setup time for the 8088 is 20 nsec. Therefore, the data read margin is 375- 20-330 = 25 nsec. Note that the output enable inputs to the ROMs have been asserted worst case 120 nsec after the beginning of T1 so that ROM address access time, not ROM output enable time, is the limiting factor in ROM access timing.

Slot Selection Generation

The HP 150 package provides two PCA slots for handling optional module PCA devices. The HP 150 architecture provides a flexible interface to the accessory slots. Accessories can be accessed through either memory or I/O access (see memory and I/O maps).

I/O addresses XX80 through XXFF have been reserved for accessory module use. This provides 128 I/O ports that can be decoded and used for processor and auxiliary device intercommunication. The most flexible arrangement that can be used for accessory module device interfacing to the 8088 is the memory mapped interface with slot select. A memory address within the 90000 - 9FFFF range causes Slot Select #1 on the front plane connecting to the external module slot #1 to be asserted and an address within A0000 - AFFFF causes Slot Select #2 to be asserted.

CAUTION

It is highly recommended that only 16K of address space be used for either of the accessory cards. The address ranges 90000-93FFF and A0000-A3FFF for slot 1 and slot 2, respectively, should be exclusively decoded and used even though the slot select lines provide 64K of address space. Accessory cards using memory space outside this range may not be compatible with any enhanced future versions of the HP 150.

What does the slot select scheme provide? An accessory PCA may choose to use or not to use the slot select line. If a device does not utilize slot select, a decode of at least the most significant four address bits must be done to detect a CPU access. More decoding of the least significant 16 address bits must be done depending on the nature of the PCA and its circuitry. In addition, the user must never plug PCAs into Slots #1 and #2 which become selected on the same address range. When fully decoding the address space, boards must be designed for a specific accessory slot.

Using the slot select lines, a more limited decode is required on the accessory PCA since the four most significant bits are decoded by the CPU in asserting the slot select line. Therefore, an accessory PCA can detect an access by decoding slot select and the appropriate subset of the least significant 16 address bits. The decode of slot select ensures that no contention will occur between accessory modules #1 an #2 since only one slot select signal will be active at one time.

The slot select thereby eliminates the need for a designer of an accessory module to be aware of all the addresses used by other existing or future accessory modules. It also limits the hardware required for decoding.

Since the accessory modules are in the 8088 address space, firmware ROMs can be placed on the accessory modules to be executed by the 8088. This way drivers for each accessory can be located in ROM on the accessory PCA and at power-on, when the terminal operating system does a logical system generation, the drivers for the modules will be used when needed for module stimulation.

Another benefit of having the accessory modules in memory space is that block transfers of data between CPU and accessory module can take place quickly with little CPU software overhead and with the enhanced flexibility of the memory access instruction set of the 8088 over the I/O instructions. Thus, a softcard CPU can be added cleanly to the system without the hardware and particularly the processor intercommunication path hampering performance.

To summarize, if a memory address between 090000H - 09FFFFH is generated by the CPU, NSLOTSEL1 is asserted. If an address between 0A0000H - 0AFFFFH is generated, NSLOTSEL2 is asserted. These signals are routed to the accessory slots with NSLOTSEL1 going to one accessory slot and NSLOTSEL2 going to the other.

CMOS RAM

256 nybbles of battery backed up CMOS RAM is located at address 0BC000H - 0BFFFFH. The 32K address space allocated to CMOS RAM images the 256 nybbles found at 0BC000H - 0BC0FFH. The nybble wide data is accessed on the lower four bits of the data bus.

CMOS DECODING AND ACCESS. Interfacing the CMOS RAM to the system required circuitry to accommodate some of the unusual address and data setup and hold timing parameters associated with the part. An address generated by the processor within CMOS memory space causes decoding circuitry to assert a CMOS address space select signal. When this signal is asserted, the processor PCA wait state generation circuitry adds 6 clock periods to the bus cycle.

CMOS POWER. The power delivered to the CMOS comes from the same power source as that used for the real time clock. The Vcc pin for the chip is maintained at approximately 5 volts when power is applied to the system and falls to about 2.5 volts when the power is turned off.

LEDs

Six LEDs are positioned on the memory PCA to report power-on test results. The LEDs are memory mapped at 0B8000H - 0BBFFFH. Any memory write within this range will access the LEDs. The LEDs are accessed on the lower six bits of the data bus.

LED DECODING. When the LED address is present, a memory write cycle is in progress, and when GO is deasserted, the LED register will be clocked with data from the CPU. The LEDs will in turn display the complement of the register contents (i.e., a "zero" turns an LED on, a "one" turns an LED off).

LED REGISTER RESET. When the power is turned on or the NRESET test point on the processor PCA is grounded, the system reset signal will reset the LED register, making all outputs go to the zero state. This will turn all the LEDs on and will indicate to the observer that +5 volts is functioning at least well enough to light the LEDs. The self-test code will then proceed to turn off the LEDs if the system tests pass; or writes an error code into the LEDs signifying the failing subsystem or component.

Dynamic RAM

The 256K of dynamic RAM requires a considerable amount of support circuitry. This section will describe the decoding, control signal generation, and RAM timing. The next section will cover the refresh mechanism used on this PCA.

DECODING. The 256K of RAM is organized as four banks of 64K. The memory space allocation is:

Bank 0	00000H - 0FFFFH	(U42 - U49)
Bank 1	10000H - 1FFFFH	(U32 - U39)
Bank 2	20000H - 2FFFFH	(U22 - U29)
Bank 3	30000H - 3FFFFH	(U12 - U19)

When a memory cycle is initiated by the 8088 (BIO/-M is low) and an address within one of the four memory regions listed above is generated, a 3-to-8 line decoder asserts one of four outputs designated for RAM bank selection. When GO becomes valid at the beginning of T2, the RAS inputs to the selcted RAM bank will all go low.

DYNAMIC RAM REFRESH. In order to retain the data that has been written into a dynamic RAM, a periodic refresh cycle must occur. The refresh cycle consists of placing a row address on the address input pins of the dynamic RAM (DRAM) and then bringing the RAS input low for a specified amount of time and then raising it back high. Each row address within the RAM must be refreshed as described within a time period specified by the RAM manufacturer in order for the data to be retained in the RAM. This memory PCA uses a dynamic RAM which requires each of its 128 row addresses to be refreshed no less than every 2 milliseconds. This specification is typical for most DRAMs. Some DRAMs have 256 row addresses and require all row addresses to be refreshed no less than every 4 msec. The refresh controller on this memory PCA is capable of meeting the requirements of both types of DRAM.

The refresh scheme used on this PCA refreshes 4 row addresses every 56 usec. Thus, all the rows are refreshed within 1.8 msec The entire refresh process takes about 2.5 usec including latencies due to the synchronizing flip-flops. Therefore, the refresh process consumes about (2.5 usec/56 usec) x 100% = 4.6% of the system bandwidth.

PCA CONFIGURATION. Normally, this PCA will be shipped with 256K of DRAM. This requires jumper W3 to be installed. If this PCA is configured with only 128K of DRAM jumpers W1 and W2 should be installed and jumper W3 should not be installed. This will allow this PCA to function properly as a 128K RAM PCA with banks 0 and 1 of RAM installed and allows an extended memory PCA in an accessory slot to have a contiguous mapping of its RAM with the RAM on this PCA starting at memory address 20000H.

ACCESSORIES SUBSYSTEM

Accessory Hardware Design Guidelines

The two accessory slots in the card cage provide the means of increasing the overall system capability by adding hardware to meet needs specific to that of certain users. The following paragraphs provide guidelines and specifications needed by designers to interface with the system hardware.

Mechanical Specifications

The accessory PCA electrically interfaces to the system through a 70 pin connector located at the front of the board. The PCA can be of the type that fits within the card cage such as the Expansion Memory PCAs. Or it may be of the type which requires interfacing to external devices and requires a special rear panel to accomodate a connector or connectors.

Mechanical drawings for the accessory card slots and rear panel are included in this section.

Power Requirements

Each of the two accessory slots have the following available power:

VOLTAGE	CURRENT
+5V	1.9A
+12V	0.45A
-12V	0.1A

If powering an external device through the power supply, the designer must use a filtering network and fuse between the external device and the HP 100 Series Personal Computer. Note that the power consumed by an accessory is limited by thermal considerations described next.

Thermal Limits

The maximum thermal dissipation allowed per accessory board is 10 watts. If there is an accessory device consuming 10W power in one slot and your board is to go in the other, you should expect temperatures on your accessory board of 20 to 30 degrees C above ambient if your board consumes 5W. If your board consumes 8.5W, you should expect to see a 20 to 40 degrees C temperature rise above ambient on your accessory board. It is best to measure the case temperatures of components on the accessory board if one wants to verify the operating temperature of a particular device is within the manufacturer's specifications.

Accessory Signal Loading Restrictions

The accessory boards have access to the signals listed in the front plane accessory slot pinout detail (table 3-5). Proper adherence to the AC and DC loading restrictions are required for per specification performance of the system. Table 3-3 lists the loading restrictions per accessory PCA.

Table 3-3. Accessory Loading Restrictions

Signal	Available Source Current	Available Sink Current	Maximum Capacitive Loading****
FPA 0-7, 12-19 ABUS 8-11	800 uA	3.5 mA	35 pF
FPD 0-7	800 uA	3.5 mA	35 pF
BIO/-M* FPNRD, FPNWRT FPNSS0, FPDT/-R FPCLK, FPGO	800 uA	3.5 mA	25 pF
NSLOTSEL n	500 uA	5.0 mA	25 pF
FPNRST	800 uA	5.0 mA	50 pF
NPFAIL	500 uA	2.0 mA	25 pF
BATV	40 uA	0	---
SHOLDA	250 uA	5.0 mA	25 pF
FULLMEM**	0	10 mA	---
NOCINT*** NDCOCINT NOCWAIT			

NOTE

* The BIO/-M signal comes from one driver and is found on pins 47 and 62 of the connector. The aggregate of the loading on the two pins must not exceed the specification in the table.

** FULLMEM, if used, must be pulled high by a pullup resistor on the accessory board.

*** These signals must be driven by an open collector driver. The pullup resistor is on the processor board and need not be on the option board. NOCWAIT requires a driver capable of sinking 18 mA (Schottky driver). The NOCINT and NDCOCINT drivers must be capable of sinking 5 mA.

**** To estimate capacitive loading, one can use 2 pF per connector pin, 5 pF/inch of PC trace, and 5-10 pF per gate terminal.

Signal Timing Diagrams

Timing diagrams in figures 3-19 thru 3-21 are based on a system with two accessory boards installed which conform to the loading restrictions stated in the previous paragraph.

Table 3-4. Accessory Slot Timing Characteristics*

Symbol	Parameter	Nom.	Min.	Max.
Tab	Clock high time		43	
Tbc	Clock low time		68	
Tac	Clock period	125		
Tal, Tbm	FPCLK to 8088 clock skew			28
Tan	BIO/-M delay from clock rising edge		14	88
Tao	FPNSS0 delay from clock rising edge		14	83
Tbp	FPA n delay from T1 beginning			102
Tbq	ABUS n delay from T1 beginning			95
Tkr	ABUS hold time from T4 rising edge		10	
Tbs	NSLOTSEL delay from T1 beginning			110
Tdt	FPGO delay from T2 beginning			48
Tju	FPGO delay from T4 beginning			48
Tdv	FPNRD delay from T2 beginning		14	128
Tjw	FPNRD delay from T4 beginning		14	108
Tvw	FPNRD width		325	
Txj	Read data setup time		49	
Tjy	Read data hold time		10	225
Taz	FPDT/-R delay from clock rising edge		14	83
TkAA	FPDT/-R delay from clock rising edge		14	83
TdBB	FPNWRT delay from T2 beginning		14	98
TjCC	FPNWRT delay from T4 beginning		14	98
TBBCC	FPNWRT width		335	
TdDD	Data valid delay from T2 beginning			84
TkEE	Data hold time from rising edge of T4		10	
TuEE	Data hold time from FPGO deassertion		30	
TFFHH, TGGII	SHOLDA delay from falling edge of clock			24
THHII	SHOLDA width (in usec)	2.5	2.25	2.63
TLLJJ, TMMKK	NOCWAIT setup time before falling edge of clock		13	
TLLMM	NOCWAIT width (in usec)	0.125		40

*All time specifications are in nsec unless stated otherwise.

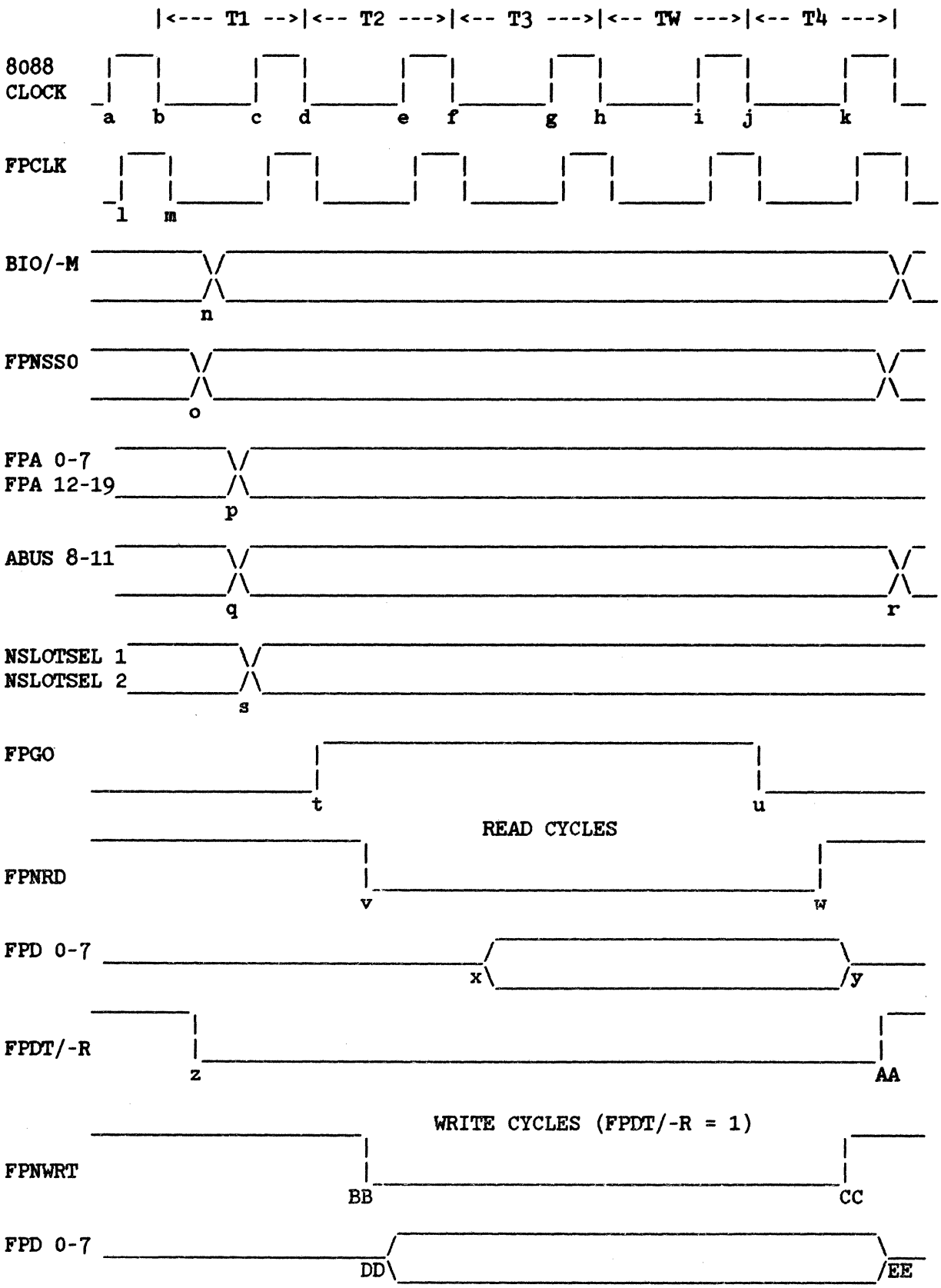


Figure 3-19. Accessory Slot Bus Cycle Timing

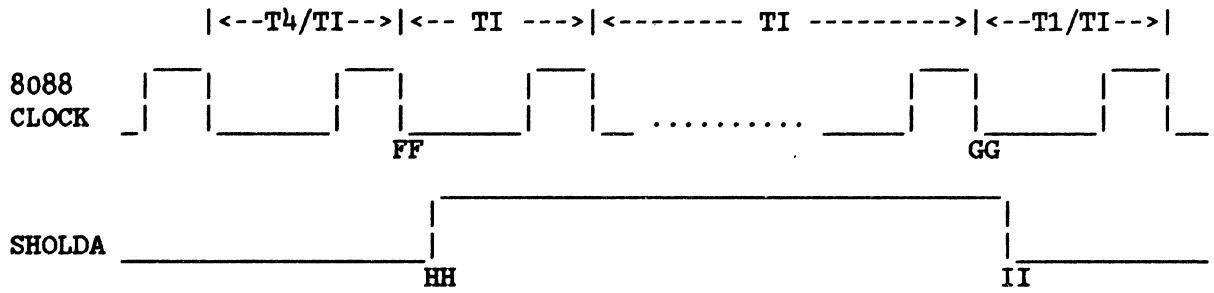


Figure 3-20. SHOLDA Timing

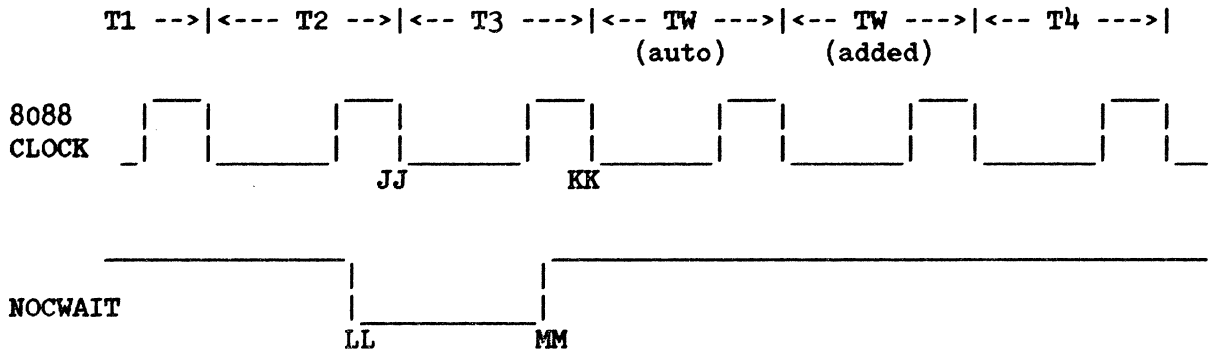


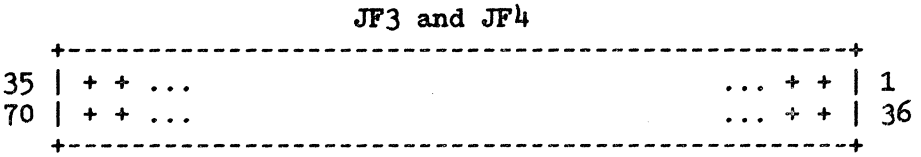
Figure 3-21. Accessory Wait State Insertion

Accessory Front Plane Connector

The accessory PCAs plug into the bottom slots of the card cage. The system signals available to the modules are listed in table 3-5.

Table 3-5. Connector Pinouts

1.	FPA 0	26.	FPD 3	51.	GND
2.	FPA 1	27.	FPD 4	52.	-12V
3.	FPA 2	28.	FPD 5	53.	SHOLDA
4.	FPA 3	29.	FPD 6	54.	FPGO
5.	FPA 4	30.	FPD 7	55.	+5V
6.	FPA 5	31.	GND	56.	GND
7.	FPA 6	32.	FPNRD	57.	FPCLK
8.	FPA 7	33.	GND	58.	(RESERVED) *
9.	ABUS 8	34.	+5V	59.	GND / FULLMEM *
10.	ABUS 9	35.	BATV	60.	+12V
11.	ABUS 10	36.	(RESERVED) *	61.	GND
12.	ABUS 11	37.	NPFAIL	62.	BIO/-M
13.	FPA 12	38.	(RESERVED) *	63.	+12V
14.	FPA 13	39.	FPNRST	64.	FPNWRT
15.	FPA 14	40.	+5V	65.	+5V
16.	FPA 15	41.	GND	66.	FPDT/-R
17.	FPA 16	42.	GND	67.	GND
18.	FPA 17	43.	NOCINT	68.	FPNSS0
19.	FPA 18	44.	GND	69.	GND
20.	FPA 19	45.	NOCWAIT	70.	NSLOTSELx **
21.	GND	46.	(RESERVED) *		
22.	GND	47.	BIO/-M		
23.	FPD 0	48.	+5V		
24.	FPD 1	49.	-12V		
25.	FPD 2	50.	NDCOCINT		



Pictorial view of option module front plane connectors (JF3 and JF4) with pin assignment (as viewed from component side of Front Plane PCA).

* Pin 59 is FULLMEM is on JF3 only. Pin 59 is GND on JF4. Connections to this pin and all pins labeled RESERVED should not be made.

** NSLOTSELx will be either NSLOTSEL1 or NSLOTSEL2 depending upon which side of the front plane the module is plugged into. The left side of the card cage (from rear view of the package) holds module 1 and gets the NSLOTSEL1 signal. The right side of the card cage holds module 2 and gets NSLOTSEL2.

Accessory Connector Signal Descriptions

The accessory connector signals described in table 3-6 reference the timing diagrams in figures 3-19 through 3-21.

Table 3-6. Signal Descriptions

FPA 0-7	The lower 8 address bits of the 20 bit address generated by the 8088. These signals are buffered and demultiplexed.
ABUS 8-11	The upper 12 address bits of the 20 bit address.
FPA 12-19	These signals also are buffered and demultiplexed.
FPD 0-7	Data bus signals from the external data bus. This data path is the means through which data is passed between the processor and the video board or option slot modules or the optional TPM.
FPCLK	Buffered 8 Mhz system clock.
FPNRD	Buffered -RD signal from the 8088 used to indicate a bus read cycle in progress.
FPNWRT	Buffered -WRT signal from the 8088 used to indicate a bus write cycle in progress.
FPNSS0	8088 bus cycle status line. The combination of FPNSS0, BIO/-M, and FPDT/-R allow boards connected to the front plane to completely decode the current bus cycle.
BIO/-M	Buffered IO/-M signal from the 8088 used to distinguish memory and I/O bus cycles.
FPDT/-R	Buffered DT/-R signal from the 8088 used to indicate direction of data from the 8088 for a given bus cycle.
FPGO	Signal which qualifies the address generated by the microprocessor. The address qualification is needed primarily for dynamic RAM circuits which cannot tolerate an assertion of RAS or CAS on a false address. FPGO goes high at the beginning of T2 and goes back low at the beginning of T4 of a bus cycle. FPGO can also be used to terminate a bus write cycle by having its falling edge used to clock data into a register or other device on an accessory device in an option slot. Terminating the writes in this way can provide better hold timing than using FPNWRT.

NOCWAIT	This line can be asserted by the option modules or video board through an open collector gate to insert wait states into a bus cycle to provide sufficient time for a bus cycle access.
NDCOCINT	This open collector interrupt signal has the same interrupt priority as the datacomm controller chip on the processor board as it shares the same input to the interrupt controller as the datacomm controller chip does. This input can be asserted by accessories for interrupt servicing.
NOCINT	System interrupt signal asserted by a device via an open collector gate. This line can be used by options slots to get processor service.
SHOLDA	Synchronized hold acknowledge signal from the 8088. HOLD is asserted by the mezzanine memory board.
FULLMEM	Indicates which mezzanine memory board option is installed. FULLMEM=0 if 128K RAM and FULLMEM=1 (it is pulled high by a pull up resistor on an installed extension memory board) if 256K RAM is on the board. An accessory should not connect to this signal pin.
NSLOTSEL1	Signal indicating an address within the 64K block allocated to option module 1 has been generated by the 8088.
NSLOTSEL2	Same as NSLOTSEL1 but asserted when address within option module 2 address space is generated.
FPNRST	Signal generated by processor board at power-on time to initialize logic circuitry.
NPFAIL	Signal generated by the power supply indicating power supply output level stability.
BATV	These two connector lines carry current from the batteries located on the video board to circuitry requiring battery back-up power such as the CMOS RAM and the CMOS real time clock.

ACCESSORY CARD HARDWARE AND ELECTRICAL

Electrical Design

HELPFUL DESIGN HINTS. Because the 8088 in the HP 150 runs in "Min" mode, the write line (FPNWR) may not rise soon enough to lock your data in. The signal FPGO is designed to allow this timing deficiency to be overcome. One way this may be implemented is by combining DT/-R with FPGO to create a write signal. The sample schematic (figure 3-31) at the end of this section shows a typical method of doing this.

Note that the signal NSLOTSEL is not guaranteed to be valid until 110 ns into the "T1" cycle. Prior to that time, glitches may be present. Edge sensitive chips should not be directly connected to this line. In general, NSLOTSEL should be combined with another of the control lines to create a valid signal. Note how NSLOTSEL is combined with FPGO on the sample schematic to create the signal NQUAL which is low only when the board is truely selected.

Based on the two points above, you have probably noticed what a handy little signal FPGO can be. FPGO is the first signal you should turn to if you come up with any timing problems in the interface to the HP 150 bus.

The HP 150 firmware has provisions to deal with options cards. To take full advantage of these features, an option card should provide an "ID BYTE" at the first address in the memory space. In the sample schematic enclosed, the "ID BYTE" has been loaded into the first address of the ROM. If a ROM is not present on the accessory card, an "ID Byte" may be generated via an octal buffer (like a LS244). Use a dip switch to set the "ID BYTE" in this case.

Be aware of the capacitive and dc loading specifications that an accessory card must meet. It doesn't take too many gates and trace lengths to exceed the specifications. Note, for example, that most control lines will not drive two Schottky gates because there would be a violation of the DC loading specification. The DC and AC loading specifications are contained in table 3-3.

GENERAL SCHEMATIC DISCUSSION. There is a "generic" accessory card schematic in figure 3-31. It includes a tranceiver interface to the data bus, buffering of the address lines, buffering and filtering of the reset line, memory and I/O space accesses, and a ROM. The memory and address reads are separated with the BIO/-M signal. When high, this signal disables the S138. The S138 determines which of the the main blocks on the accessory card are selected. If an I/O access occurs to the address set by the dip switches, then the state of the F74 flip flop is toggled. This will select one of the two LEDs. The "controller" is a piece of LSI logic which takes care of the external peripheral device.

Another feature that is nice to have, although not present on the enclosed schematic, is a software hard reset of the accessory card. This allows the card to be hard reset without a hard reset of the system.

TRANSCEIVER SCHEMATIC DISCUSSION. The schematic in figure 3-22 includes a tranceiver on the data bus. This circuit illustrates how a "general purpose" tranceiver might be put on the data bus.

There are a number of problems in putting a tranceiver on the data bus including:

1. Creating a write pulse which will clock data into its destination and allow sufficient hold time before the tranceiver is disabled.

2. Ensuring that the tranceiver is not disabled prior to the data hold time on a read.

3. Ensuring that the tranceiver is disabled prior to DT/-R changing state.

The circuit proposed should solve these problems without causing contention on the HP 150's data bus or on the accessory card's data bus. In addition, it does not rely on unspecified minimum delay times. Check the timing to see if this circuit will work for a particular application.

The circuit works by first latching (with the rising edge of FPCLK) a board select signal (NQUAL) as the tranceiver enable signal. This delays the tranceiver enable signal, allowing read/write cycles to finish prior to disabling the tranceiver.

Since the tranceiver enable signal has been delayed, DT/-R must also be delayed. The second flip-flop latches the state of DT/-R with the falling edge of FPCLK. This delay allows the tranceiver to be disabled prior to a direction change (at the end of an instruction cycle) while still allowing the direction to be established prior to tranceiver enable (at the start of an instruction cycle).

Of course, the circuits above will not be useful for every design but they may be helpful in saving a little time in the design process. Remember that the ultimate responsibility is yours.

Parts List

1 FAST '00
1 FAST '74
1 ALS '245
1 LS '04

DC Loading (For Tranceiver Circuit Only)

FPDT/-R	40 uA High 1.2 mA Low
FPGO	40 uA High 1.2 mA Low
FPCLK	40 uA High 1.0 mA Low
NSLOTSEL	20 uA High .4 mA Low

References

1982 Fast Data Book (Fairchild)
1981 TTL Data Book (Texas Instruments)
1983 ALS/AS Data Book (Texas Instruments)

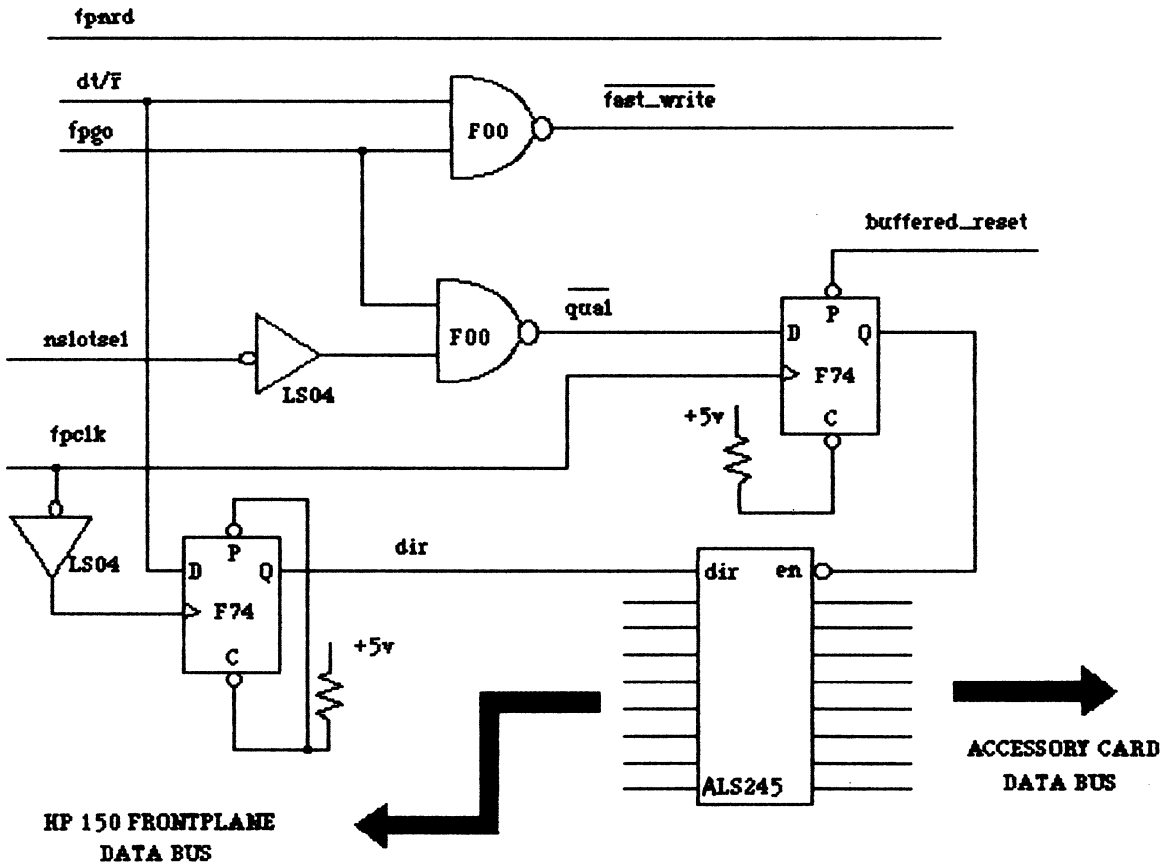


Figure 3-22. Tranceiver Schematic

Mechanical Design

HELPFUL DESIGN HINTS. Consider the height restriction drawing since the tabs where the I/O panel thumbscrews attach can interfere with components on the accessory card.

The I/O Panel should be of a sturdy design so it will transfer sufficient force to cause the frontplane connector to seat. When testing the mechanical design, remove the metal chassis and watch the connector as the thumbscrews are tightened. It is possible to design a panel such that the thumbscrews will go all the way in, yet the frontplane connector will not move.

Make sure that the I/O panel has as many holes for airflow as possible. The air will flow along the long dimension of the accessory board. For most effective use of this airflow, lay the chips out with their long dimension perpendicular to this airflow. Also, lay your "hot" chips as close to the I/O panel as possible.

Be sure to mask off the annular ring around the thumbscrew's hole and to use an inside star washer between the thumbscrew and the I/O panel. This will ensure that there is a solid ground path between the I/O panel and the chassis of the HP 150.

I/O Panels with a connector should have the ground plane split near the I/O panel to bypass the connector shield to the I/O Panel ground. This will prevent high frequency noise (like ESD) from entering the system.

I/O PANEL DESIGN. Accessory cards not requiring an I/O Panel should be based on the board blank drawing titled "Board Blank - no I/O Panel" with two extractors and two extractor retainer pins to remove the accessory card from the HP 150. Drawings for the extractors as well as the extractor retainer pins are provided in this manual.

If an I/O panel is required, all the required dimensions may be obtained from the drawings enclosed. Please note that the accessory card fits under the L-bracket for the design presented. The I/O Panel should be solid and sturdy since the force from the thumbscrews must be transmitted to the frontplane connector in order to fully insert the connector.

LIST OF VENDORS

CAUTION

Parts may be available from the following vendors. HP does not guarantee or warrant any of the following parts or vendors.

- **Frontplane Connector**

  Vendor: Amp
  Harrisburg, Pa.

  Vendor Part Number: 1-102584-0

- **Thumbscrew**

  Vendor: Precision Metal
  3402 Enterprise Ave.
  Hayward, Ca 94545
  (415) 887-1401

  Vendor Part Number: 45641-00001-2 (also HP Part No.)

Use the drawing in figure 3-24 to set this part up with its own part number. This will prevent any possible changes to the part from affecting you. The drawing will also allow the part to be set up with any vendor you wish.

- **E-Ring Retainer**

  Vendor: Truarc Retaining Rings Division
  Waldes Kohinoor, Inc.
  47-16 Austel Place
  Long Island City, NY 11101
  (212) 392-3100

  Contact the factory for the distributor near you.

  Vendor part number: Truarc X-5133-11 H.

- **L-Bracket**

  Vendor: Pike Tool and Grinding Inc.
  4205 High Country Road
  Colorado Springs, Colo. 80907
  (303) 598-9611

  Vendor Part Number: 45641-00001-1 (also HP Part No.)

Use the drawing in figure 3-25 to set this part up with its own part number. This will prevent any possible changes to the part from affecting you. Again, the drawing is provided for this part if you wish to set it up yourself.

- **Accessory Card Extractor** (figure 3-26)

  Vendor: Trend Plastic
  1245 Laurelwood Rd.
  Santa Clara, CA. 95050
  (408) 727-5797

  Vendor Part Number: 0403-0473 (also HP Part No.)

- **Accessory Card Extractor Retainer Pin** (figure 3-27)

  Vendor: Drivelock, Inc.
  Sycamore, IL.
  (815) 895-8161

  Vendor Part Number: 9945

  One possible distributor for this part is:

  Bearing Engineers, Inc.
  1250 Space Park Way
  Mountain View, CA. 94040
  (415) 969-1155

  The Bearing part number is 3-0.330-R-SP.

I/O PANEL PAINT SPECIFICATIONS

Scope

This specification establishes the requirements for pigment composition, to achieve a color in HP's Corporate Color Program.

Requirements

- o Pigments used to manufacture a color should be from the approved Hewlett-Packard List, "Pigments Used to Achieve the Corporate Color".

- o Pigment substitutes may be required for specific type of material that uses higher temperature, (such as plastic) to be produced in a Corporate Color.

  a. Use pigment with equivalent physical and performance properties as recommended on the pigment list.

  b. Required physical and performance properties of pigments:

     1. Heat Resistance
     2. Bleeding Characteristics
     3. Chemical Resistance
     4. Exposure Performance (Ultra-violet,etc.)

Quantitative

- o Pigment Composition (approximation - adjust % for tinting purposes)

  a. Paint Pigments for Parchment White Case, Visual Color Standard No. 6009-0130:

Trace %	Lampblack
18.55 %	Titanium Dioxide White
.31 %	Yellow Iron Oxide
Trace %	Red Iron Oxide
%	

  Gloss Standard: 18+-3 Gloss Units at 85 degree /-

  b. Paint Pigments for Cobblestone Gray Accent, Visual Color Standard No. 6009-0133:

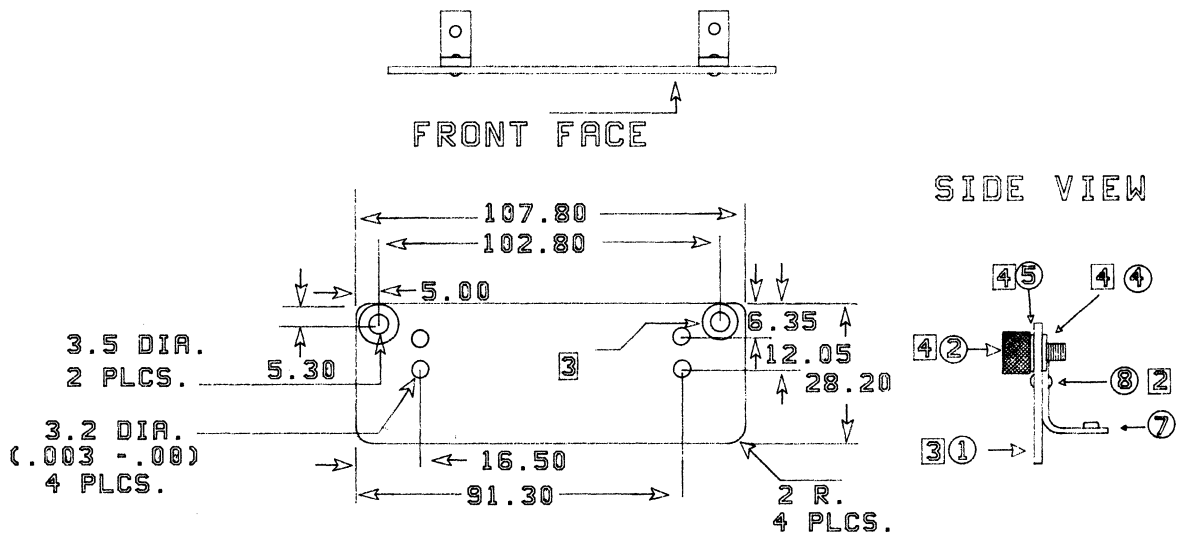
8.0 %	Lampblack
28.4 %	Titanium Dioxide White
10.9 %	Yellow Iron Oxide
4.5 %	Red Iron Oxide
48.3 %	Other

  Gloss Standard: 18+-3 Gloss Units at 85 degree /-

Drawings

Included in this section are the following drawings:

- I/O Panel
- Retainer Thumbscrew
- L-Bracket
- Accessory Card Extractor
- Accessory Card Extractor Pin
- Board Blank With I/O Panel (Accessory Card Details)
- Height Restriction (Accessory Card Clearances)
- Board Blank - No I/O Panel
- Accessory Card Schematic



NOTES

1. FINISH: SAND TO DEBURR, ZINC CHROMATE CONVERSION .005MM THICK.
2. RIVET ITEM 7 TO ITEM 1 WITH ITEM 8 (2 PLCS.)
3. MASK OFF TWO AREAS OF 8MM DIA. AT THE TWO 3.5 DIA. HOLES AND SPRAY PAINT THE FRONT SIDE AND EDGES OF ITEM 1.
4. INSTALL ITEMS 2, 4, 5 AS SHOWN ON SIDE VIEW

ITEM	QTY.	MATERIAL-DESCRIPTION	MAT'L-PART NO.	MAT'L-DWG. NO.	MAT'L-SPEC.
8	2	RIVETS .125" X 7/32	CHICAGO RIVET + MACHINE CO. R-3682 OR EQUIV		
7	2	P.C. MOUNTING BRACKET		45641-00001-1	
6	A/R	PAINT WATER BASE, COBBLE. GRAY			
5	2	INSIDE STAR WASHER	FOR #8 SCREW		
4	2	E-RING RETAINING RING	TRUARC x-5133-11H OR EQUIV.		
3	A/R	PAINT WATER BASE, PARCH. WHITE			
2	2	THUMB KNOB		45641-00001-2	
1	A/R	COLD ROLLED STEEL 1.2 mm TK			

Figure 3-23. I/O Panel

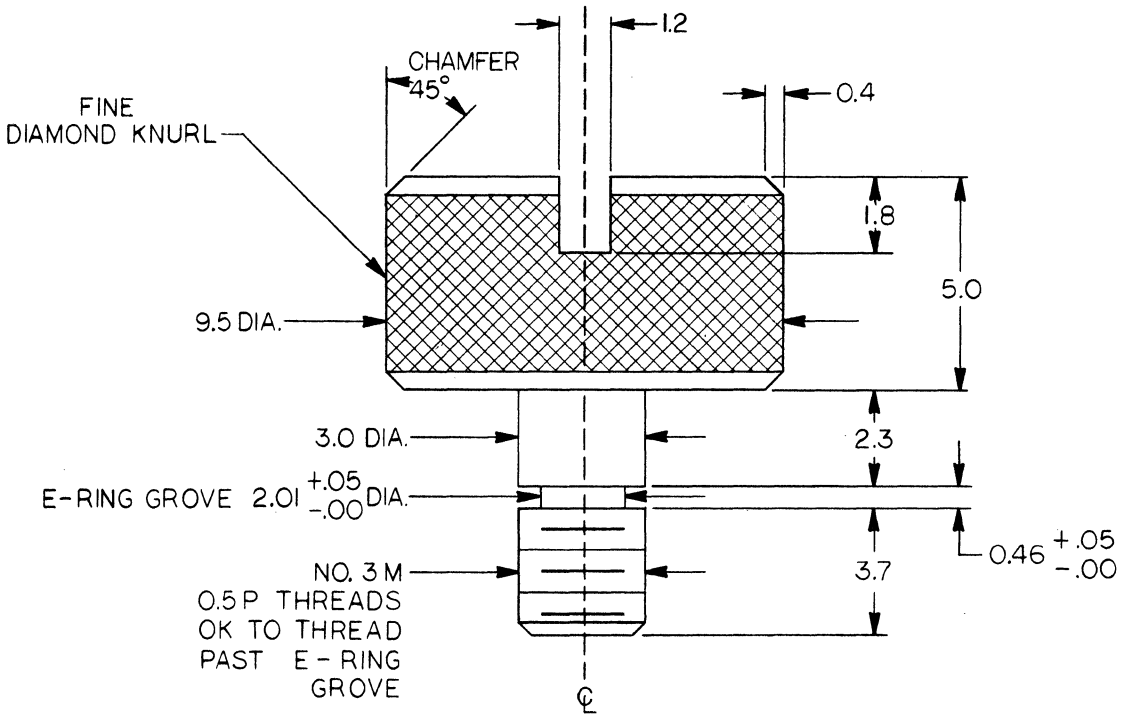


Figure 3-24. Retainer Thumbscrew

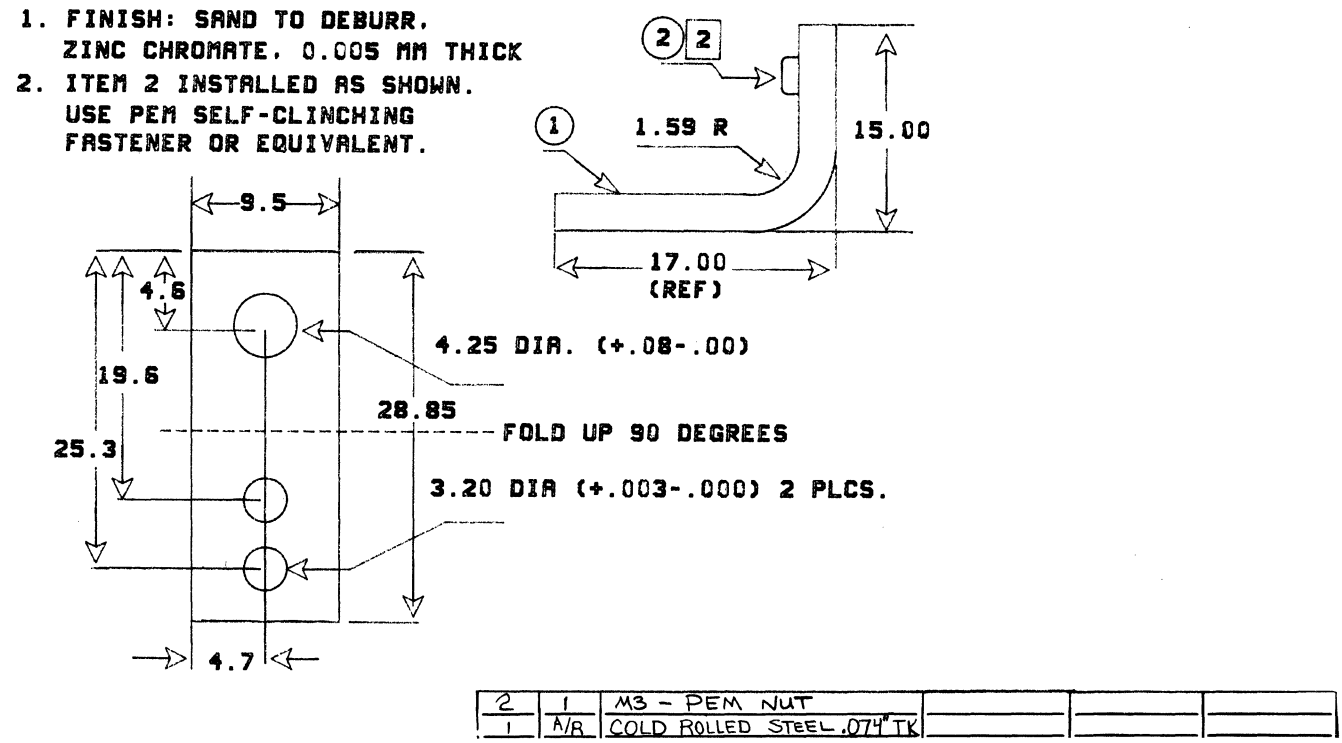


Figure 3-25. L-Bracket

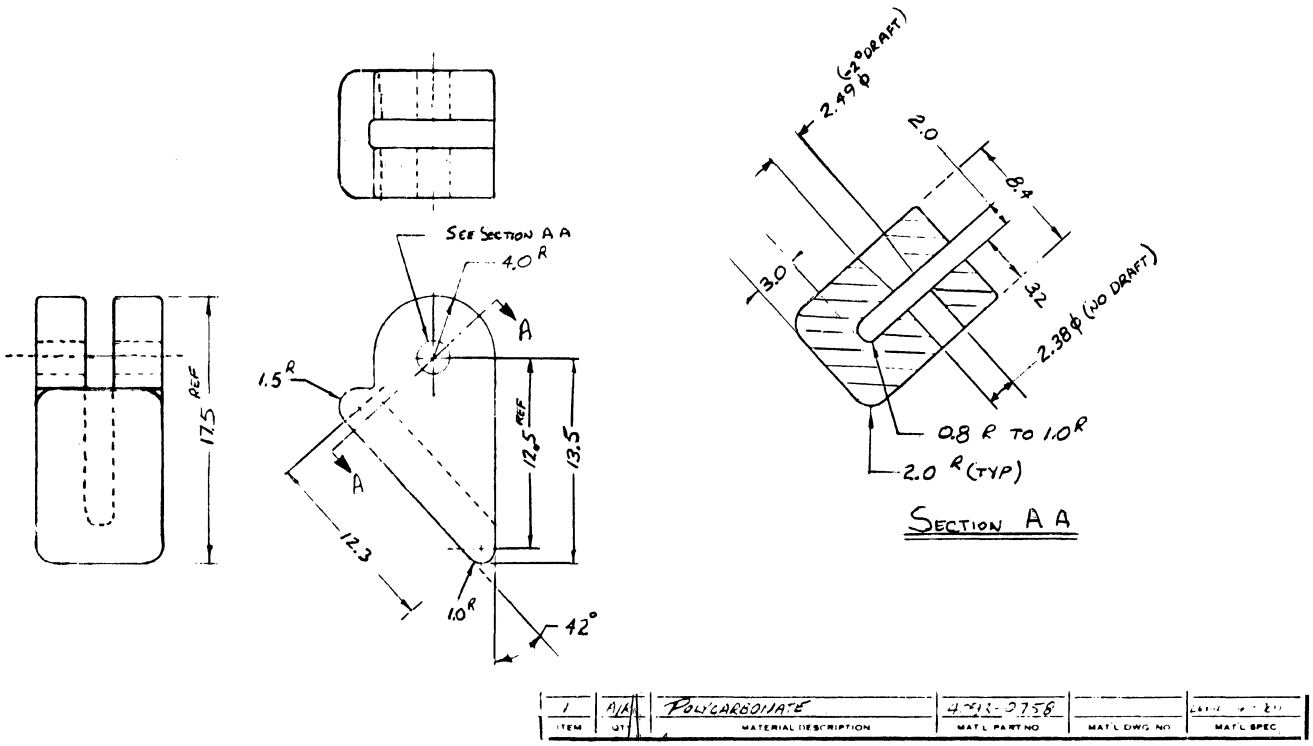


Figure 3-26. Accessory Card Extractor

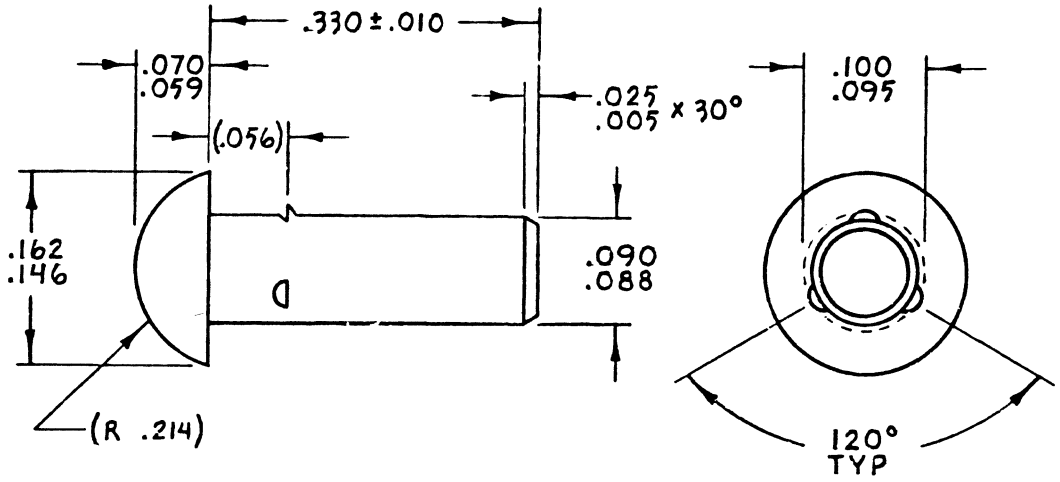


Figure 3-27. Accessory Card Extractor Retainer Pin

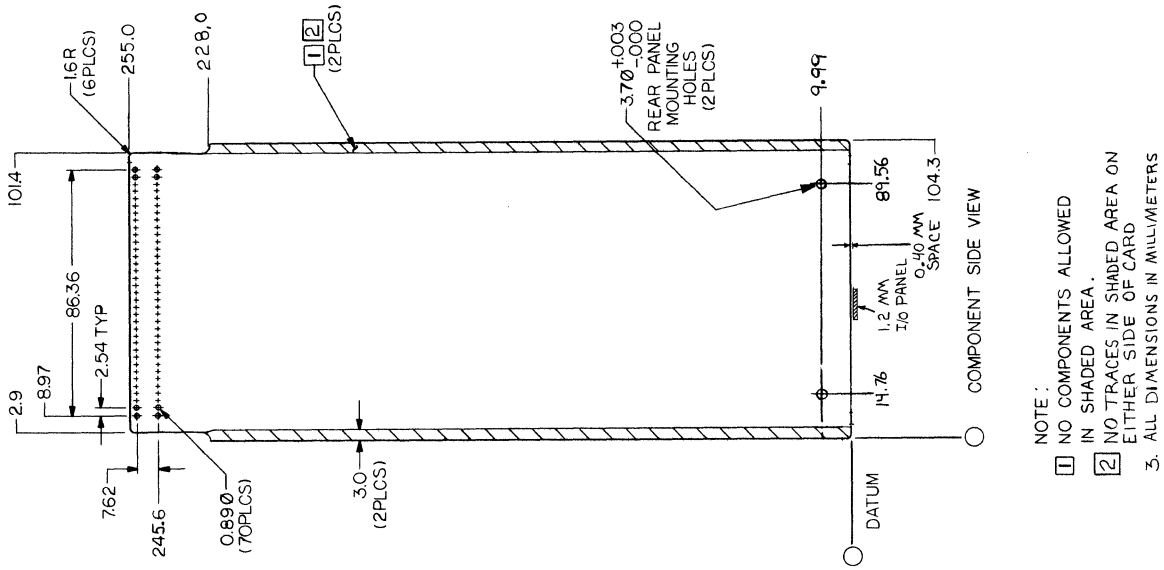


Figure 3-28. Board Blank Drawing With I/O Panel (Accessory Card Details)

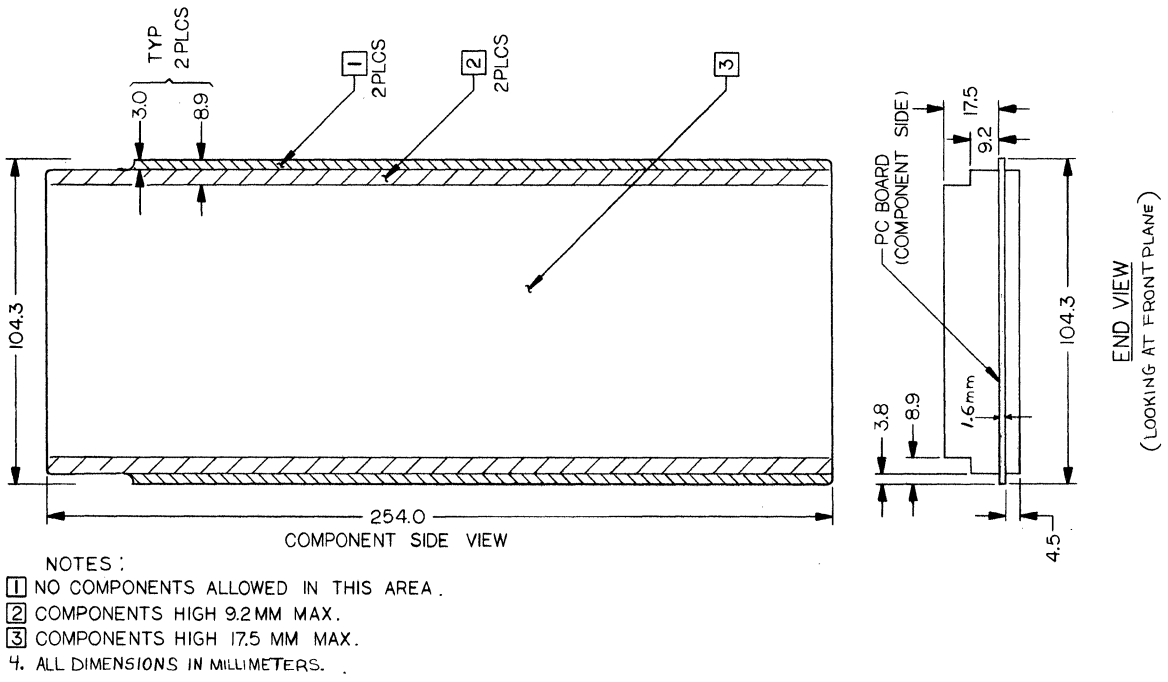
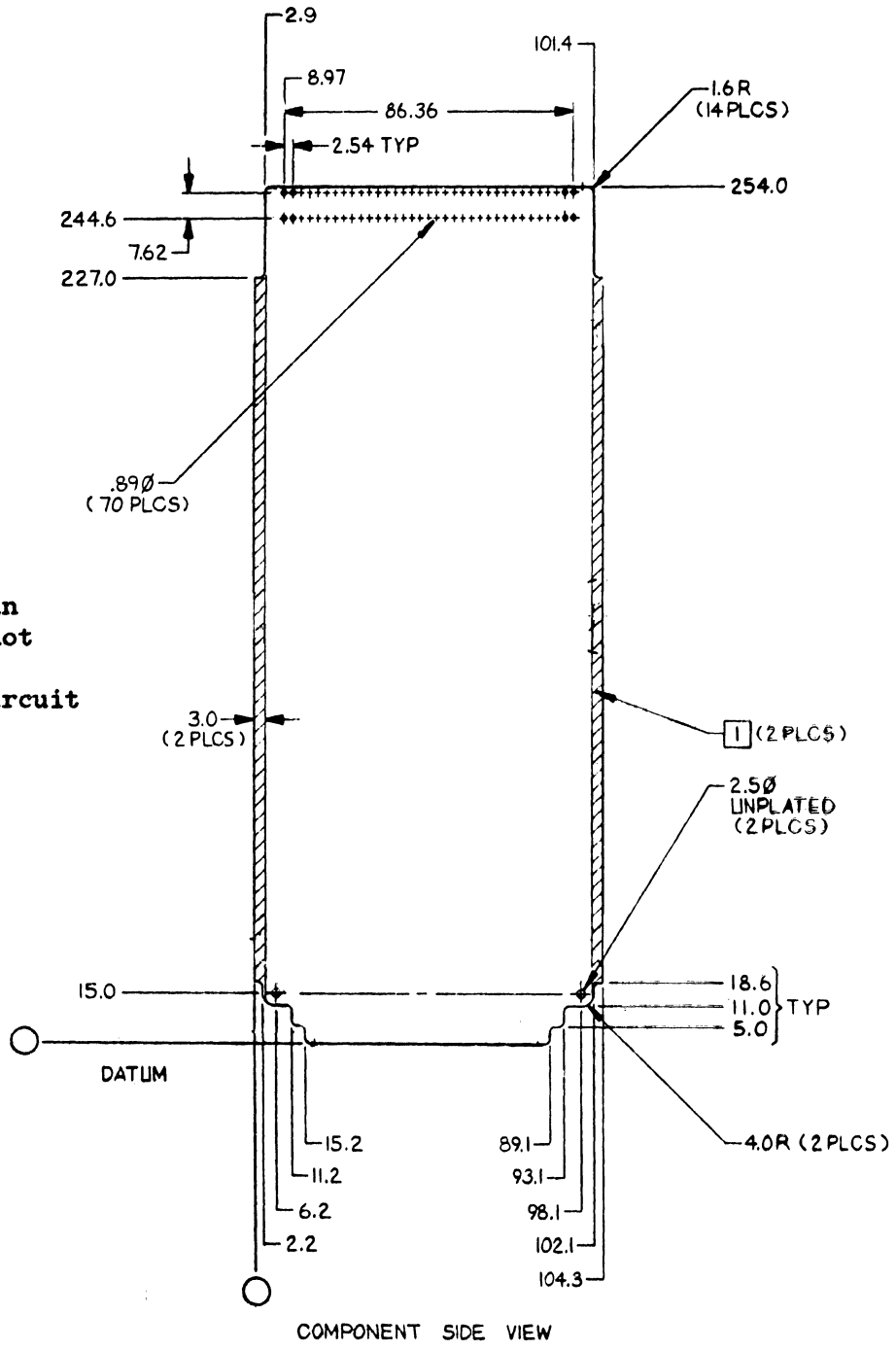


Figure 3-29. Height Restriction Drawing (Accessory Card Clearances)



NOTE

No components allowed in shaded area. Also do not run any traces in the shaded region on the circuit or component side.

Figure 3-30. Board Blank - No I/O Panel

This is page is blank.

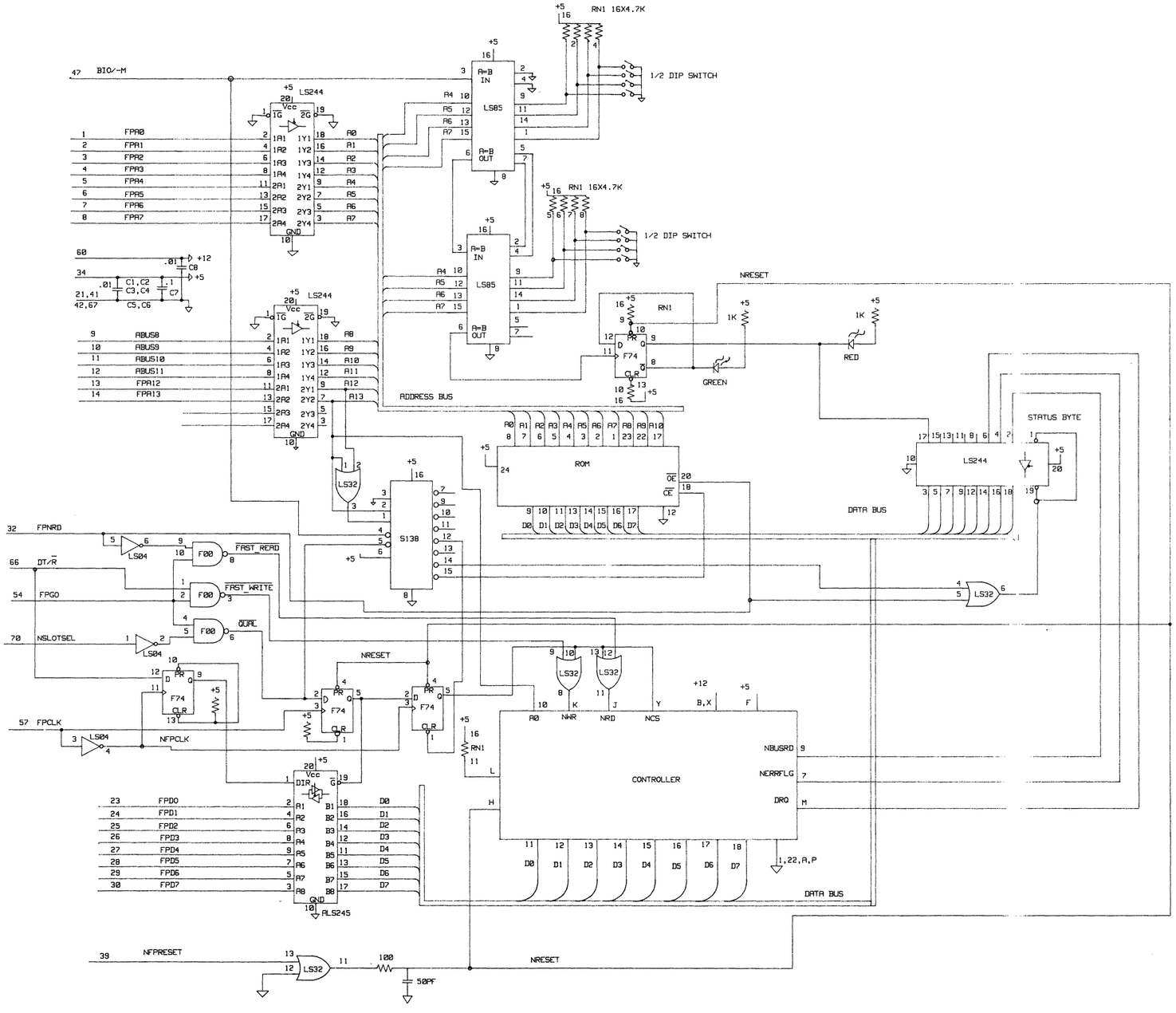


Figure 3-31. Typical Accessory Card Schematic

MEMORY AND I/O MAPPING

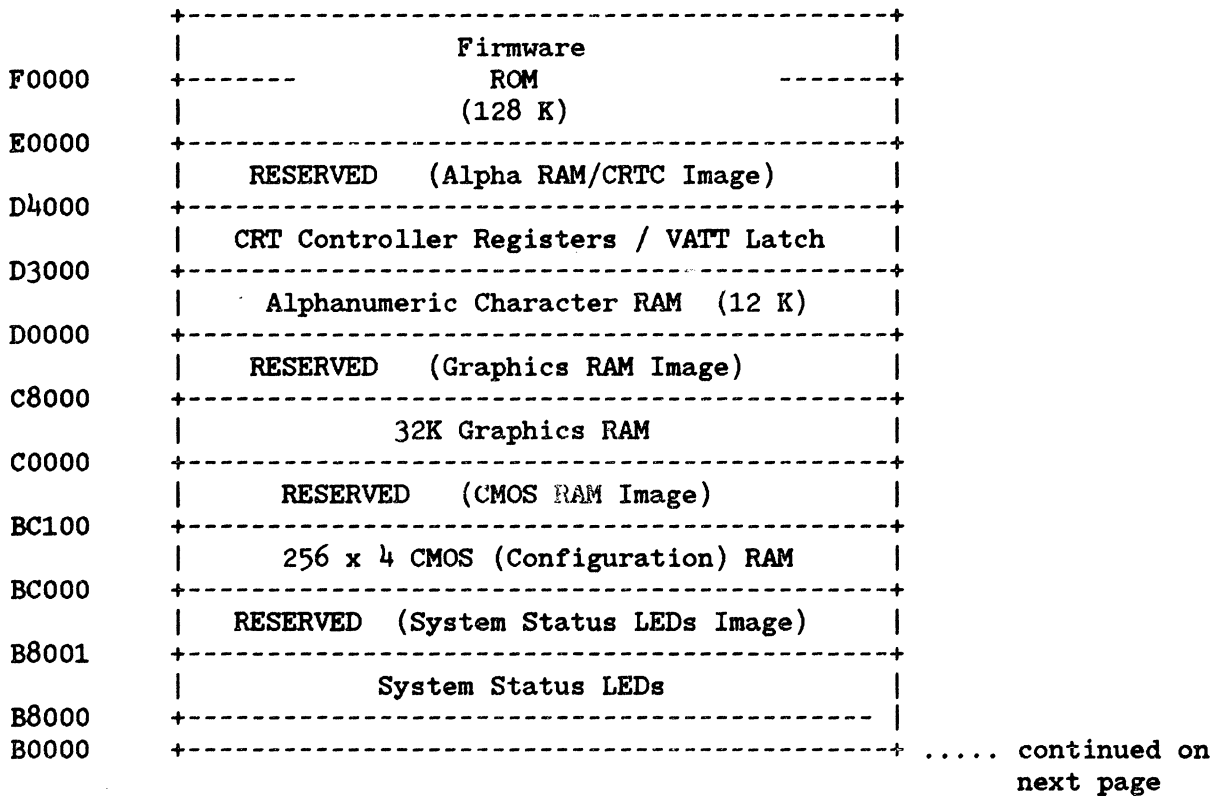
SECTION 4

This section contains memory and input/output maps for the HP 150 memory and input/output address spaces. Included is information describing the register and bit mappings of all devices which consume a portion of the 8088 processor's address space for both memory and input/output instruction types. More information regarding the functionality of the devices themselves may be found in Section 3, Hardware Subsystems.

CONTENTS

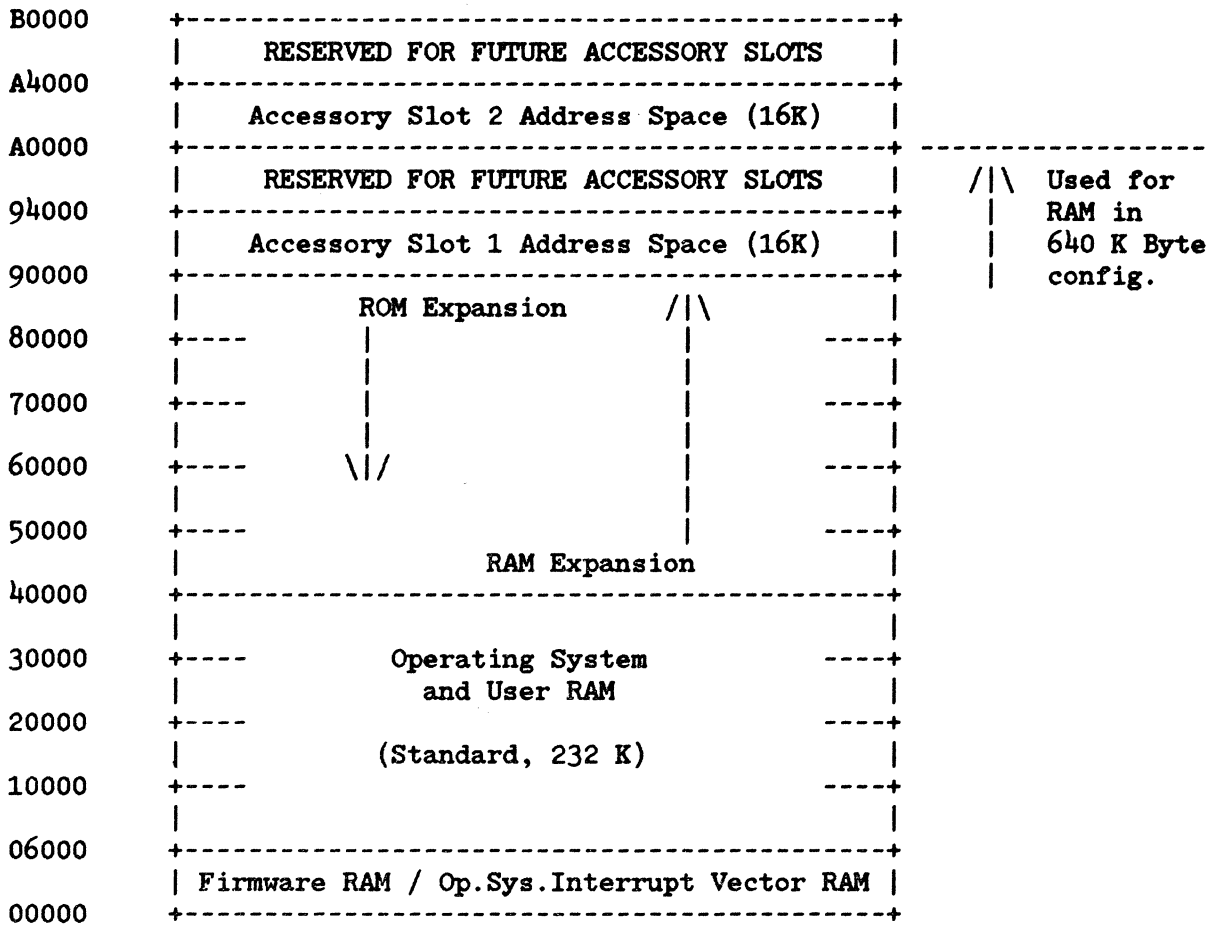
MEMORY MAPPED DEVICES

HP 150 Memory Map



..... continued on next page

HP 150 Memory Map continued



NOTE

For a more detailed map of Operating System, User, and Firmware RAM useage, refer to System Software (Section 5), Operating System Memory Map.

CRT Controller Registers

Register Type / Bit Definition D7	D6	D5	D4	D3	D2	D1	D0	Register #	Initialization Register Data (hex)	8088-Addr (hex)	Read/Write
MSB	CHARS./HORIZONTAL PERIOD						LSB	R0 #	73	D3000	WRITE
MSB	CHARACTERS/ DATA ROW						LSB	R1 #	*4F	D3002	WRITE
MSB	HORIZONTAL DELAY						LSB	R2 #	2E	D3004	WRITE
MSB	HORIZONTAL SYNC WIDTH						LSB	R3 #	07	D3006	WRITE
MSB	VERTICAL SYNC WIDTH						LSB	R4 #	13	D3008	WRITE
MSB	VERTICAL DELAY						LSB	R5 #	20	D300A	WRITE
PIN CONFG	CURSOR SKEW			BLANK SKEW				R6 #	C9	D300C	WRITE
MSB	VISIBLE DATA ROWS/ FRAME						LSB	R7 #	*1A #	D300E	WRITE
SCAN LINES/FRM (B10)	(B8)	SCAN LINES/DATA ROW (MSB)					(LSB)	R8 #	2D	D3010	WRITE
(B7)	SCAN LINES / FRAME						(B0)	R9 #	9F	D3012	WRITE
DMA DISABLE	DMA BURST DLY MSB		LSB	DMA BURST COUNT MSB			LSB	RA #	70	D3014	WRITE
X	PB/SS	INTERLACE MODES		OPERATION MODES			2XC/1XC	RB #	41	D3016	WRITE
MSB	TABLE START REGISTER(LS BYTE)						LSB	RC #	91	D3018	WRITE
ADDRESS MODE	TABLE START REG (MS BYTE) (MSB)						(LSB)	RD #	97	D301A	WRITE
MSB	AUX. ADDRESS REG.1 (LS BYTE)						LSB	RE #	FF	D301C	WRITE
ROW ATTR'S	AUX ADDR REG 1(MS BYTE) (MSB)						(LSB)	RF #	3F	D301E	WRITE
MSB	SEQUENTIAL BREAK REG. 1						LSB	R10 +	FF	D3020	WRITE
MSB	DATA ROW START REGISTER						LSB	R11	FF	D3022	WRITE
MSB	DATA ROW END/SEQU. BRK REG 2						LSB	R12	FF	D3024	WRITE
MSB	AUX. ADDRESS REG.2 (LS BYTE)						LSB	R13 +	00	D3026	WRITE
ROW ATTR'S	AUX ADDRESS REG 2 (MS BYTE) (MSB)						(LSB)	R14 +	00	D3028	WRITE
START COMMAND								R15	00	D302A	READ/WRITE
RESET COMMAND								R16 +	00	D302C	READ/WRITE
OFST OVR-FLOW	OFFSET VALUE (MSB)					(LSB)	0	R17	00	D302E	WRITE
VERTICAL CURSOR REGISTER (ROW COORD.) (MSB)							(LSB)	R18	FF	D3030	WRITE
								R38 +	--	D3070	READ
HORIZONTAL CURSOR REG. (COL. COORD.) (MSB)							(LSB)	R19	00	D3032	WRITE
								R39 +	--	D3072	READ
	VERT RE-TRCE	LGHT PEN	INTERRUPT ENABLE REG X	X	X	X	FRM TIM-ER	R1A #	40	D3034	WRITE
INT PEN-DING	VERT RE-TRCE	LGHT PEN	STATUS REGISTER		ODD/EVEN	X	FRM TMER	R3A	--	D3074	READ
MSB	VERT LIGHT PEN REG(ROW COORD)						LSB	R3B +	--	D3076	READ
MSB	HOR LIGHT PEN REG (COL COORD)						LSB	R3C +	--	D3078	READ
		VIDEO ATTRIBUTES REGISTR									
X	X	X	DISP ON		CURS BLRT	BLRT	BLOB GREN	"VATT"	00	D30BE	WRITE

* These registers are programmed with N -1.
\# These registers must have these initialization values.
\+ These registers may not be accessible on future HP 150 revisions and must not be written to or read from.

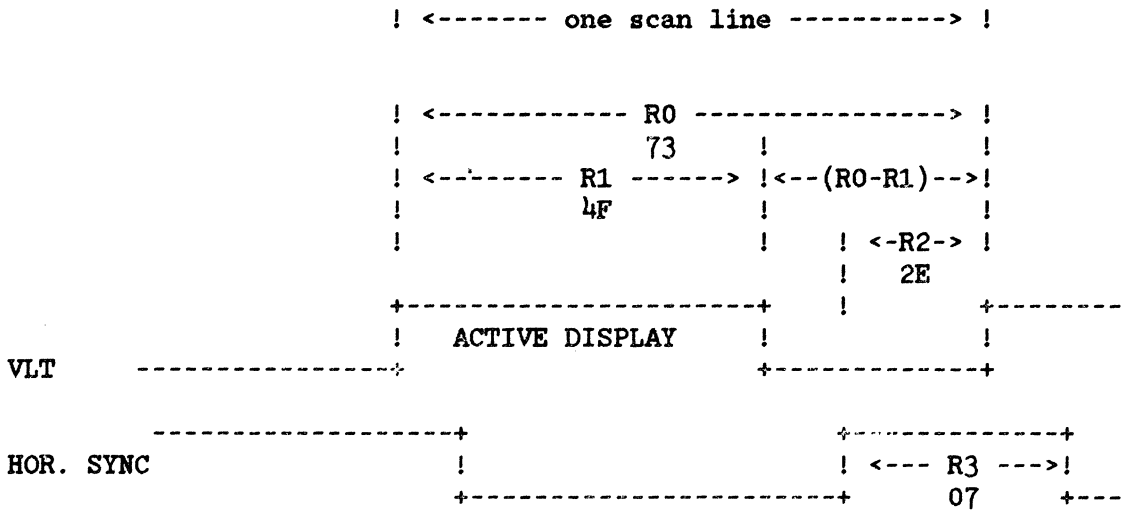
NOTE

For a full description of the CRT Controller registers, refer to the SMC9007 data in Standard Microsystems Corporation 1982 Data Book. Use of registers noted as being not accessible on future HP 150 revisions is not recommended and such access may result in software incompatibilities with certain HP 150 units. All addresses not specifically shown are reserved and should not be used.

A description of CRT controller registers as those registers are used and supported by the HP 150 follows.

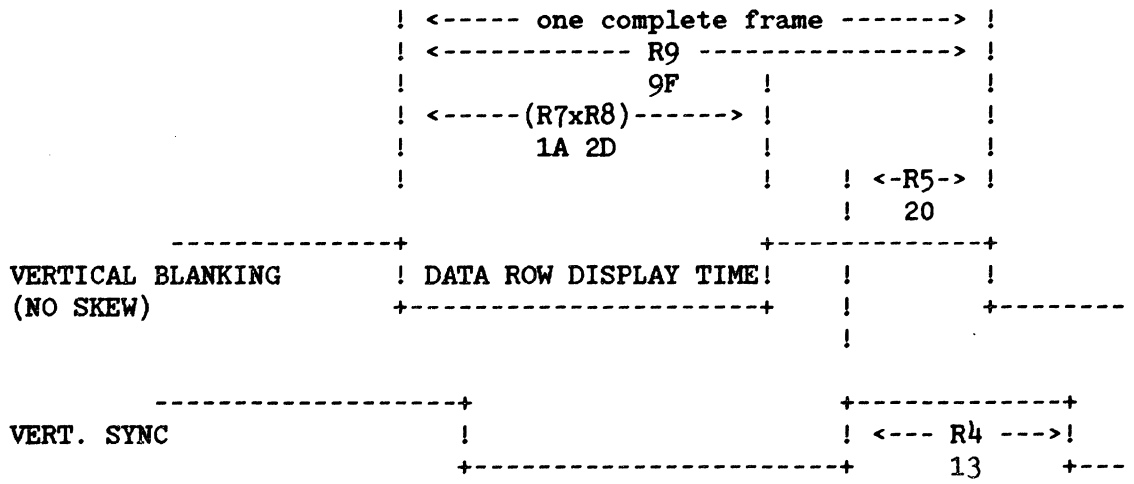
HORIZONTAL TIMING REGISTERS (R0, R1, R2 AND R3)

These registers define the horizontal scan line timing. They must contain the (hexadecimal) values shown.



VERTICAL TIMING REGISTERS (R4, R5, R7, R8 AND R9)

These registers define the vertical frame timing. They must contain the (hexadecimal) values shown.



PIN CONFIGURATION/SKEW BITS REGISTER (R6)

- Contains pin configuration information (bit 7,6)
- Cursor skew (bits 5,4,3) define the number of character clocks the cursor signal is delayed from VLT
- Blank skew (bits 2,1,0) define the number of character clocks the horizontal blank component of the CBLANK signal is delayed from VLT.
- Both cursor skew and blank skew are the value "001" for the 1 character skew.
- Must contain C9 Hex.

DMA CONTROL REGISTER (RA)

- DMA disable (bit 7). A logic "1" on this pin forces the SMC9007 DMA request into the inactive state, and the address bus will enter its high impedance state.
- DMA burst delay (bits 6,5,4). This register is loaded with "111" for zero delay, allowing all characters to be retreived from video RAM in one burst.
- DMA Burst Count (bits 3,2,1,0). Not Used.
- Must contain 70 Hex.

CONTROL REGISTER (RB)

- 7 bit register
- Smooth scroll mechanism is enabled by writing a "1" to bit 6.
- Interlace, (bits 5,4). "00" - non interlaced mode.
- Operation mode (bits 3,2,1). "000" for repetitive memory addressing.
- Single/double height cursor (bit 0). "1" = single.
- Must contain 41 Hex.

TABLE START REGISTER (RC AND RD)

- These registers point to the address where the row table begins.
- The registers are set up the following way for contiguous row table mode:
  - * register D (bits 7,6) = "10"
  - * register D (bits 5-0) = upper 6 bits of the 14 bit address
  - * register C (bits 7-0) = lower 8 bits of the 14 bit address.
- RC must contain 91 Hex, RD must contain 97 Hex.

AUXILIARY REGISTER 1 (RE AND RF)

- Not used, except for bits 7,6 in register F which must be "00" for single height, single width characters.
- RE must contain FF Hex, RF must contain 3FH.

SEQUENTIAL BREAK REGISTER 1 (R10)

- This register may not be accessible on certain HP 150 revisions and must not be used.

DATA ROW START REGISTER (R11)

- Defines the first data row number at which a smooth scroll operation begins.
- Is initialized by the system firmware to FF Hex.

DATA ROW END REGISTER (R12)

- The row numerically one less than the row defined by this register is the last data row on which a smooth scroll will occur.
- Is initialized by the system firmware to FF Hex.

AUXILIARY ADDRESS REGISTER 2 (R13 AND R14)

- This register may not be accessible on certain HP 150 revisions and must not be used.

START COMMAND (R15)

- During initialization of the SMC9007, after all vital screen parameters are loaded, a start command can be initiated by addressing this dummy register location.

RESET COMMAND (R16)

- This register may not be accessible on certain HP 150 revisions and must not be used.

SMOOTH SCROLL OFFSET REGISTER (R17)

- This register is loaded with the scan line offset number to allow a smooth scroll operation to occur. The offset register causes the scan line counter to start at the programmed value rather than zero for the data row that starts the smooth scroll interval.
- Must have data bit 7 (most significant) cleared (0).

VERTICAL CURSOR REGISTER (R18)

- This register specifies the data row in which the cursor appears.

HORIZONTAL CURSOR REGISTER (R19)

- This register specifies the character position in which the cursor appears.

CURSOR REGISTERS R38 AND R39 (READ)

- These registers may not be accessible on some revisions of the HP 150 and must not be used.

INTERRUPT ENABLE REGISTER (R1A)

- This 3 bit write only register allows each of the 3 SMC9007 interrupts to be enabled or disabled.
- Bit 6 is set to "1" to enable vertical retrace interrupts.
- Must contain 40 Hex.

STATUS REGISTER (R3A)

- This register is only used to clear the interrupt bit. The value read is irrelevent.

VERTICAL LIGHT PEN REGISTER (R3B)

- This register may not be accessible on certain revisions of the HP 150 and must not be used.

HORIZONTAL LIGHT PEN REGISTER (R3C)

- This register may not be accessible on certain revisions of the HP 150 and must not be used.

VIDEO ATTRIBUTE LATCH

This latch stores 5 bits of video data, and can be updated during the blank portions of the video frame. It is a memory mapped I/O register with the 8088 address D30BE hexadecimal. The register is actually decoded as a non-existent SMC9007 register.

VIDEO ATTRIBUTE LATCH PIN DEFINITION

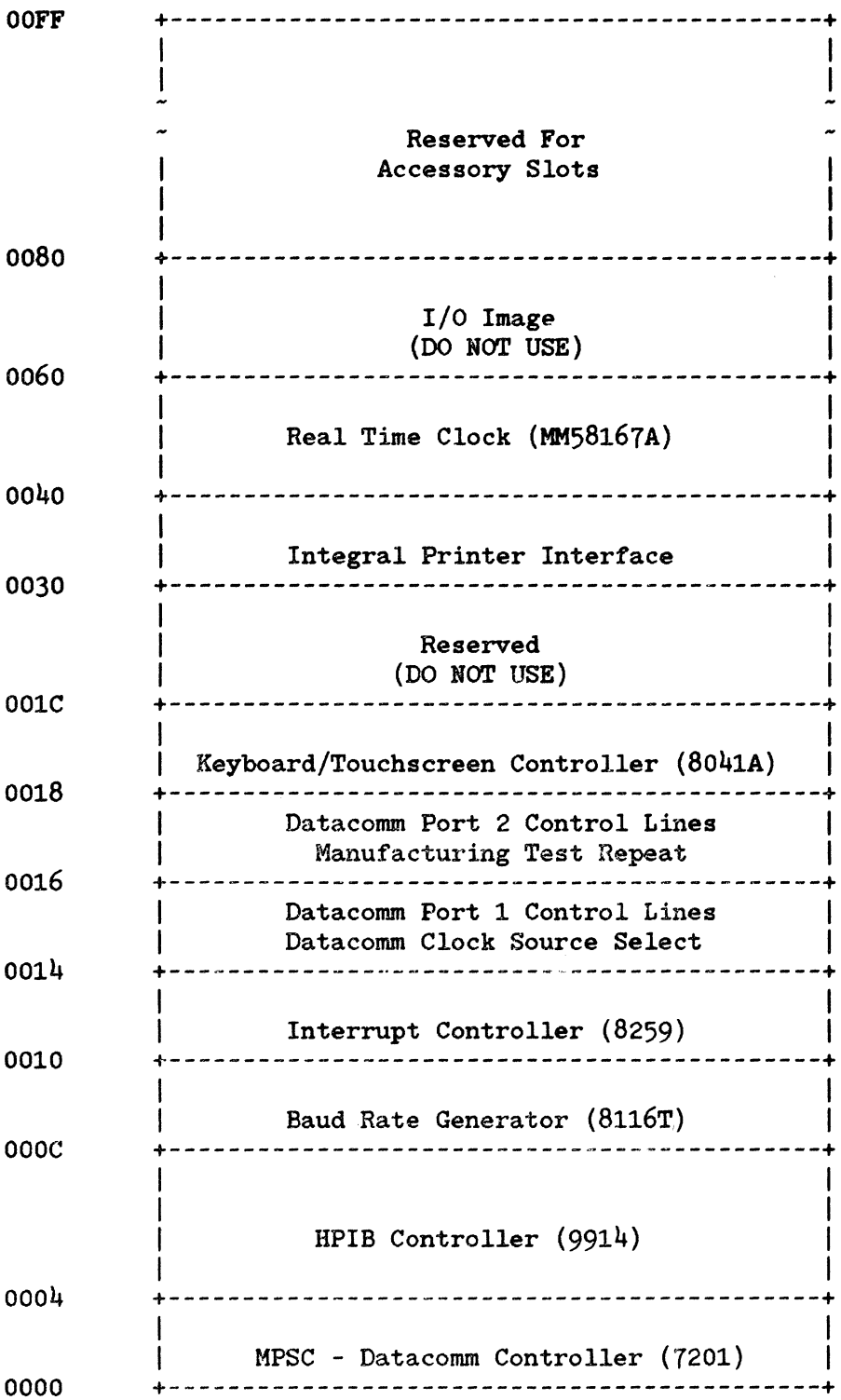
DISPON	BLRT	CBLRT	BLOB	GREN
D4	D3	D2	D1	D0

note: - positive logic is used (1 = on, 0 = off)
- when writing to this register, D5-D7 are don't cares

DISPON	:	Alpha Display On
BLRT	:	Character Blink Rate Characters with BL (blink) enhancement set will be blanked on the screen while BLRT is a "1."
CBLRT	:	Cursor Blink Rate The character for which CURS from the SMC9007 is active high, the cursor will be blinked when CBLRT is a "1."
BLOB	:	Blob Cursor Select (otherwise double scan line cursor)
GREN	:	Graphics Display Enable

I/O MAPPED DEVICES

HP 150 I/O Map



NOTE

I/O mapped devices at 0000 Hex through 0080 Hex are not exclusively decoded to 16 bits. Rather they may be accessed through any combination of high order byte bits. For future hardware compatibility it is highly recommended that programs which access I/O mapped devices directly at these addresses do so with a high order byte address of 00 Hex. Accessory cards may exclusively decode a 16 bit address. See the accessory card specifications for more information.

Real Time Clock (MM58167A)

Address	Register	Type
0056-005FH	Not Used	reserved
0055H	"GO" Comamand Register	write
0054H	Status Bit	read
0053H	RAM Register Reset	write
0052H	Counter Register Reset	write
0051H	Interrupt Control Register	write
0050H	Interrupt Status Register	read
004FH	Month RAM Register	read/write
004EH	Date RAM Register	read/write
004DH	Day RAM Register	read/write
004CH	Hour RAM Register	read/write
004BH	Minutes RAM Register	read/write
004AH	Seconds RAM Register	read/write
0049H	Tenth/Hundreth Sec RAM Reg	read/write
0048H	Millisecond RAM Register	read/write
0047H	Month Counter Register	read/write
0046H	Date Counter Register	read/write
0045H	Day Counter Register	read/write
0044H	Hour Counter Register	read/write
0043H	Minutes Counter Register	read/write
0042H	Seconds Counter Register	read/write
0041H	Tenth/Hundreth Sec Counter	read/write
0040H	Millisecond Counter Register	read/write

For more information on this part, see the MM58167A Data Sheet, National Semiconductor Corporation.

Integral Printer Interface

Address	Register	Type
0030H	Status/Data	read/write

Bit:	7	6	5	4	3	2	1	0
READ	-	-	-	-	-	NPAPER OUT	ONLINE	ACK
WRITE	D7	D6	D5	D4	D3	D2	D1	D0

Keyboard / Touchscreen Controller (8041A)

Address	Register	Type
0019H	Status / Command Register	read/write
0018H	Data Register	read/write

For more information on the use of these registers, see the "Hardware Subsystems" section, "Keyboard and Touchscreen".

Datacomm Port 2 Control Lines / Manuf Test Repeat

Address	Register	Type
0016H	OCR1/OCD1, OCR2/OCD2, DM Ctrl. Lines, Port 1 detect, Manufac. test bit	read/write

Bit:	7	6	5	4	3	2	1	0
READ	----	----	----	POD	DM	MTST	OCR2	OCR1
WRITE	----	----	----	----	----	----	OCD2	OCD1

In the above:

POD = 0 if Port 1 datacomm PCA is not present.
POD = 1 if Port 1 datacomm PCA is in place.
MTST = 0 if jumper wire is not grounding U62 pin 14.
MTST = 1 if U62 pin 14 is grounded causing repetition of manufacturing test segment that failed.

Datacomm Port 1 Control Lines / Clock Source Select

Address	Register	Type
0014H	OCR1/OCD1, OCR2/OCD2, DM Ctrl. Lines, Clock Source Select	read/write

Bit:	7	6	5	4	3	2	1	0
READ	0	0	0	1	DM	0	OCR2	OCR1
WRITE	Clock Source		----	----	----	----	OCD2	OCD1

Clock source select bits are defined as follows:

Bits 7	Bits 6	Receive Clock	Transmit Clock
0	0	x16	x16
0	1	x1	x1
1	0	RT	ST
1	1	RT	ST

Interrupt Controller (8259A)

Address	Register	Type
0010,0011H	Interrupt Controller Registers	read/write

For more information on this part, see the 8259A Data Sheet, Intel Corporation.

Baud Rate Generator (8116T)

Address	Register	Type
000CH	Baud Rate Select	write

Baud Rate Select Coding:

Port 2 Control D7 D6 D5 D4	Port 1 Control D3 D2 D1 D0	Clock Output (Hz.)	Baud Rate
0 0 0 0	0 0 0 0	800	50 *
0 0 0 1	0 0 0 1	1200	75 *
0 0 1 0	0 0 1 0	1760	110
0 0 1 1	0 0 1 1	2152	134.5 *
0 1 0 0	0 1 0 0	2400	150
0 1 0 1	0 1 0 1	4800	300
0 1 1 0	0 1 1 0	9600	600
0 1 1 1	0 1 1 1	19200	1200
1 0 0 0	1 0 0 0	28800	1800 *
1 0 0 1	1 0 0 1	32000	2000 *
1 0 1 0	1 0 1 0	38400	2400
1 0 1 1	1 0 1 1	57600	3600 *
1 1 0 0	1 1 0 0	76800	4800
1 1 0 1	1 1 0 1	115200	7200 *
1 1 1 0	1 1 1 0	153600	9600
1 1 1 1	1 1 1 1	307200	19200

* Denotes baud rates not configurable through the HP 150 Config menus.

HPIB Controller (9914)

Address	Register	Type
000BH	Command Pass Thru /Parallel Poll	read/write
000AH	Address Status Register	read
0009H	Address Switch / Address Reg	read/write
0008H	Interrupt Status 0/Interrupt Mask 0	read/write
0007H	Data In / Data Out	read/write
0006H	Bus Status / Auxiliary Command	read/write
0005H	Serial Poll register	read
0004H	Interrupt Status 1/Interrupt Mask 1	read/write

For more information on this part, see the TMS9914 Data Sheet, Texas Instruments Incorporated.

MPSC - Datacomm Controller (7201/8274)

Address	Register	Type
0003H	Channel B Control Reg	read/write
0002H	Channel B Data Reg	read/write
0001H	Channel A Control Reg	read/write
0000H	Channel A Data Reg	read/write

For more information on programming this part, see the "PD7201 Multiprotocol Serial Communications Controller Technical Manual", NEC Electronics U.S.A. Incorporated, or 8274 Data Sheet, Intel Corporation.

SYSTEM SOFTWARE

SECTION 5

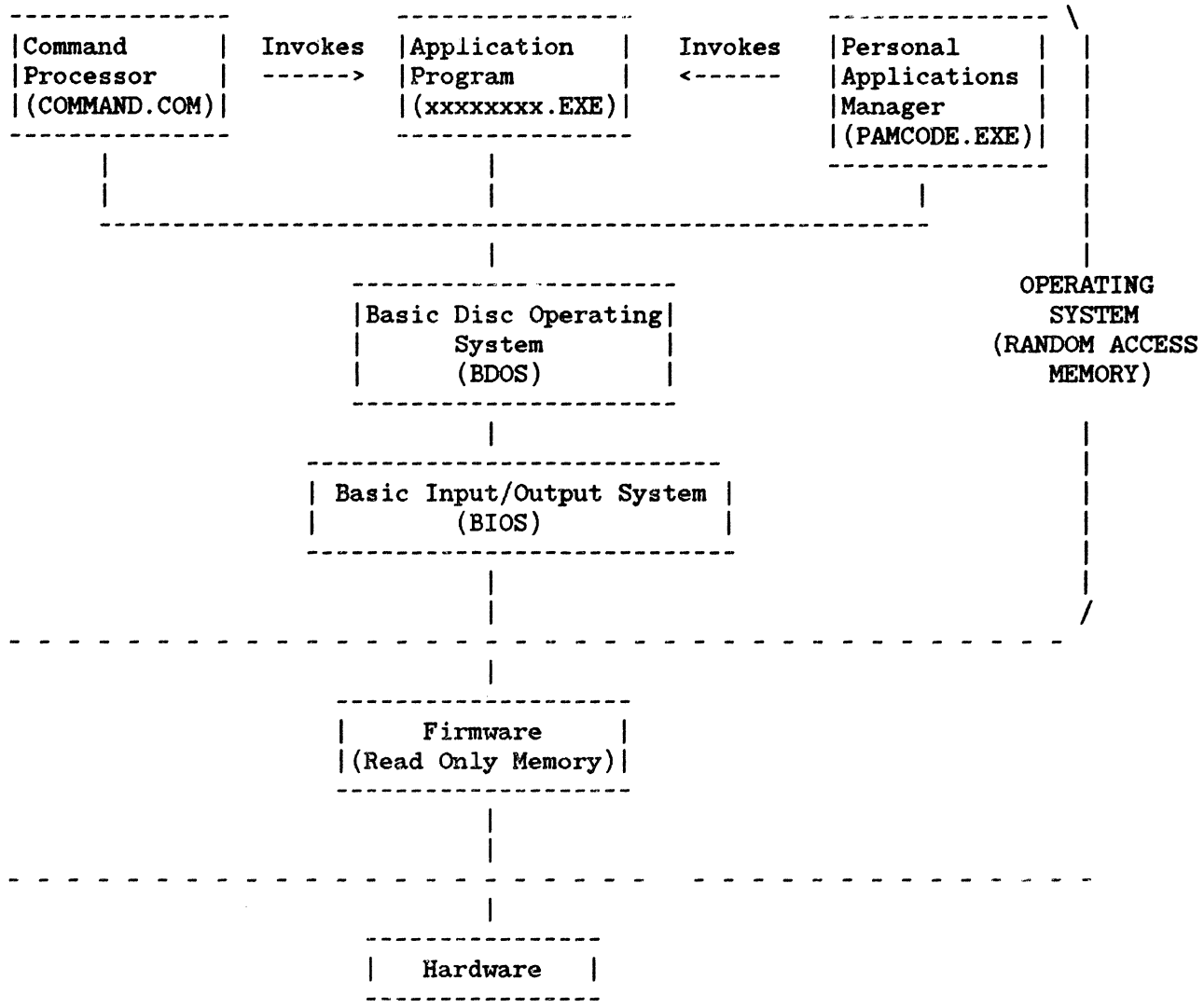
The software environment of the HP 150 is covered in this section. Included is a discussion of the operating system, MS-DOS, and HP's enhancements to it, installable and BIOS devices, and the disc format and directory structure including drive specific capacity and organizational information.

CONTENTS

OPERATING SYSTEM STRUCTURE

This chapter deals with the operating system on the HP 150, MS-DOS from Microsoft Corporation, and its interface with the HP 150 hardware and firmware.

The operating system can be partitioned into several layers:



The Command Processor

The Command Processor is a program file called COMMAND.COM.
When running under MS-DOS COMMANDS mode, this program acts as the user interface to the system. Its function is to read, parse, and execute command line inputs such as "COPY A:*.* B:". Command lines may come from the keyboard or from a batch file on disc.

The Command Processor contains a number of built-in commands such as COPY, DIR, DEL, and REN. That is, these commands are recognized and acted upon directly by the COMMAND.COM program. Given an input line the command processor first checks for a built-in command. If the input line was not a directive for one of the built-in commands, the first word of the command line is checked against filenames with the extensions .COM, .EXE, and .BAT (in that order) on the default disc drive. Files with extensions .COM or .EXE are executable programs. .COM files are single group, non-relocatable programs. .EXE files are relocatable. .BAT files are text files containing one or more command lines which are processed sequentially by the Command Processor.

The Command Processor consists of three parts: the resident, initialization, and transient parts. The resident part handles all standard MSDOS errors including the CONTROL-C and Fatal Error Abort interrupts, plus it contains the code necessary to reload its transient part if necessary. The initialization portion contains the code necessary to process the AUTOEXEC file and determine the segment at which programs are to be loaded. This part of the command module is overlaid by the first program it loads in. The transient part of the command module contains all of the internal command processors and the batch file processor. This portion of the command module displays the disc prompt, reads commands from the keyboard or batchfile, and processes them.

The Personal Applications Manager (P.A.M.)

P.A.M. is similar in structure to the MS-DOS Command Processor, COMMAND.COM. However, rather than being driven by MS-DOS command line-type inputs, the Personal Applications Manager presents a friendly face to the user through touchscreen menus. Together with the File Manager, it offers essentially all the functionality of
the generic MS-DOS command processor, however it is much easier for the novice user to operate. The CONFIG.SYS file indicates to
MS-DOS that it should load and execute PAMCODE.EXE rather than COMMAND.COM on the HP 150. See the CONFIG.SYS File and System Booting sections for more information.

Application Programs

Application programs are generally program files with the extension .EXE. Both the Command Processor and Personal Applications Manager can invoke application programs.

Basic Disc Operating System (BDOS)

The Basic Disc Operating System is the heart of the MS-DOS operating system. The BDOS is supplied by Microsoft Corporation and is standard across a wide variety of equipment from many vendors. BDOS presents a system-independent hardware interface for the Command Processor and applications programs. It is for this reason that applications programs written to run under MS-DOS are to a certain extent easily transported from machine to machine.

The BDOS exists as a file called MSDOS.SYS residing on all MS-DOS system discs. It is a collection of system management and input/output functions termed System Functions.

Application programs interface with the BDOS through System Function calls. These calls are made by loading 8088 registers on the HP 150 with parameters including a function number identifier, and then issuing a software interrupt.

An application program written to perform all I/O and System Management functions through the documented BDOS System Functions will be truly machine independent, and capable of being run on other machines utilizing a compatible version of MS-DOS. However many I/O intensive applications bypass BDOS for the sake of efficiency, commonly accessing lower-levels of the operating system including the Basic Input/Output System (BIOS) and even the hardware. At this point, machine dependencies and non-portability is introduced.

For details of System Functions see the HP 150 Programmers Reference Manual.

Basic Input/Output System (BIOS)

The Basic Input/Output System (BIOS) processes BDOS I/O requests. While the BDOS knows nothing of the details of the specific piece of the hardware upon which it runs, the BIOS is very knowledgeable. MS-DOS is customized to run on the HP 150 through the BIOS.

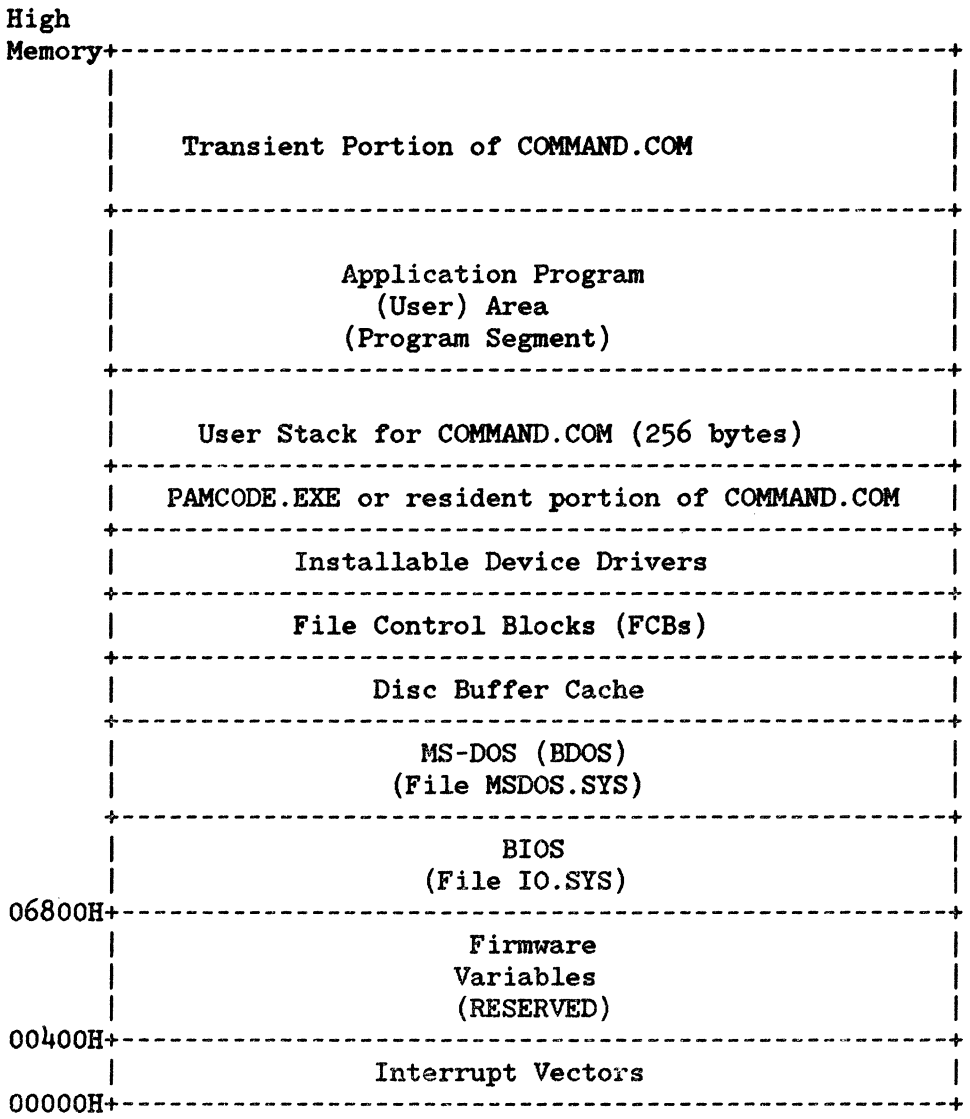
Just as the BDOS exists on system discs as a file named MSDOS.SYS, the BIOS is contained in a file called IO.SYS. These two files are read into HP 150 RAM memory during system initialization (booting) and much of them remains in memory to perform I/O and System Management functions on behalf of applications programs.

MS-DOS (the BDOS) invokes the BIOS for each I/O request by passing the BIOS a long pointer (20 bit address) to an I/O packet. This is true for all requests whether they are from MS-DOS itself or on behalf of its applications. The packet is a self-contained structure which contains all the information required to fully execute the I/O request. For more information on this interface, see "Calling a Device Driver" in this section.

The BIOS is discussed in more detail later in this chapter.

OPERATING SYSTEM MEMORY USAGE

Operating System Memory Map



Interrupt Vectors

Interrupts within the HP 150 are triggered by hardware attached to the 8088 microprocessor, software interrupt instructions performed by MS-DOS, software interrupt instructions from application (user) programs, or under some special circumstances, by the 8088 itself.

Every interrupt is assigned a type code that identifies it to the 8088. Interrupts are identified as "INT n" where n (the type code) is a number between 0 and 255 inclusive. The type code is used by the 8088 to calculate a location in the memory based interrupt vector table containing the four byte address of the interrupt routine. The interrupt vector for INT 0 is at address 00000H, the vector for INT 1 is at address 00004H, and so on. The interrupt vector table can contain up to 256 vectors, one for each interrupt type.

Each entry in the table is a doubleword pointer containing the address of the procedure that is to service interrupts of that type. The higher addressed word of the pointer contains the base address of the segment containing the procedure. The lower addressed word contains the procedure's offset from the beginning of the segment. Since each entry is four bytes long, the 8088 can calculate the location of the correct entry for a given interrupt type by simply multiplying the type by four. For more information on how the 8088 processor treats interrupts and the conditions which cause Intel-reserved interrupts, see the IAPX86/88, 186/188 User's Manual, Programmer's Reference, Intel Corporation, May 1983.

The 256 interrupt types are pre-allocated for the HP 150. The following table describes that allocation.

Starting Address	Interrupt Vector Description	Usage
003FCH	Type 255 Pointer - (**** not used ****)	/|\
.	.	|
.	.	|
.	.	Available
.	.	|
00200H	Type 128 Pointer - (**** not used ****)	|/
001FCH	Type 127 Pointer - (RESERVED)	
.	.	Reserved
.	.	for HP
00120H	Type 72 Pointer - (RESERVED)	
0011CH	Type 71 Pointer - HP 150 Hardware (IR7)	/|\
00118H	Type 70 Pointer - HP 150 Hardware (IR6)	|
00114H	Type 69 Pointer - HP 150 Hardware (IR5)	Reserved
00110H	Type 68 Pointer - HP 150 Hardware (IR4)	for and
0010CH	Type 67 Pointer - HP 150 Hardware (IR3)	used by HP
00108H	Type 66 Pointer - HP 150 Hardware (IR2)	hardware.
00104H	Type 65 Pointer - HP 150 Hardware (IR1)	|
00100H	Type 64 Pointer - HP 150 Hardware (IR0)	|/
000FCH	Type 63 Pointer - (**** not used ****)	/|\
.		|
.		|
.		|
000A8H	Type 42 Pointer - (**** not used ****)	|
000A4H	Type 41 Pointer - MSDOS Interrupt 29	Reserved
000A0H	Type 40 Pointer - MSDOS Interrupt 28	for
0009CH	Type 39 Pointer - MSDOS Interrupt 27	MicroSoft
00098H	Type 38 Pointer - MSDOS Interrupt 26	|
00094H	Type 37 Pointer - MSDOS Interrupt 25	|
00090H	Type 36 Pointer - MSDOS Interrupt 24	|
0008CH	Type 35 Pointer - MSDOS Interrupt 23	|
00088H	Type 34 Pointer - MSDOS Interrupt 22	|
00084H	Type 33 Pointer - MSDOS Interrupt 21	|
00080H	Type 32 Pointer - MSDOS Interrupt 20	|/
0007CH	Type 31 Pointer - (**** not used ****)	/|\
.		|
.		|
.		|
00014H	Type 5 Pointer . (**** not used ****)	Reserved
00010H	Type 4 Pointer - Intel Dedicated	for
0000CH	Type 3 Pointer - Intel Dedicated	Intel
00008H	Type 2 Pointer - Intel Dedicated	|
00004H	Type 1 Pointer - Intel Dedicated	|
00000H	Type 0 Pointer - Intel Dedicated	|/

MS-DOS INTERRUPTS

Interrupt 27H: Terminate But Stay Resident
Interrupt 26H: Absolute Disc Write
Interrupt 25H: Absolute Disc Read
Interrupt 24H: Fatal Error Abort Address
Interrupt 23H: CONTROL-C Exit Address
Interrupt 22H: Terminate Address
Interrupt 21H: Function Request
Interrupt 20H: Program Terminate

HP 150 HARDWARE INTERRUPTS

IR7: Real Time Clock (MM58167A)
IR6: Not Used (Tied High)
IR5: HPIB Controller (9914)
IR4: *Integral Printer,Accessory Slot NOCINT (Low Priority Open Collector)
IR3: Keyboard and Touchscreen
IR2: Not Used (Tied High)
IR1: *MPSC (Datacomm Controller),Accy. Slot NDCOCINT (High Priority O/C)
IR0: Video Controller (9007)

* These interrupts may be initiated from an accessory board by using the NOCINT and NDOCOINT open collector lines. Either may be used, IR1 has higher priority than IR4 (IR1 will be serviced prior to IR2-IR7). See Accessory Board Subsystem in the Hardware Subsystems section of this manual.

Firmware Variables

The firmware uses the RAM space between 00400H and 067FFH for jump vectors and other working data storage. See Section 6, "System Firmware" for more information.

BIOS and BDOS

The Basic Input/Output System (BIOS) and the Basic Disc Operating System (BDOS) are loaded from disc during initialization. BDOS is loaded following BIOS loading and initialization, and overlays the initialization portion of BIOS. The origin address of MS-DOS is BIOS revision dependent.

Disc Buffer Cache

The Disc Buffer Cache is an area used for buffering of disc data for all drives. It includes portions or all of the File Allocation Tables (depending upon cache size), the root directory, and data for non-sector oriented reads and writes. The size of the Disc Buffer Cache may be

altered from the default by the "BUFFERS=<number>" command in the CONFIG.SYS file. See the CONFIG.SYS file information in this section for a description of the "BUFFERS" command.

File Control Blocks (FCBs)

File control blocks contain file information taken from the directory, and include pointers marking an access position.
Two types of FCBs exist. An unopened FCB is one that contains only a drive specifier and a filename, which can contain wild card characters (* and ?). An opened FCB contains all fields filled by the MS-DOS 'Open File' system function call.

Space for two FCBs is allocated in the Program Segment Prefix (in the application program area). Space is allocated above the Disc Buffer Cache for a number of additional FCBs. The "FILES=<number>" command in the CONFIG.SYS file can be used to alter the amount of space allocated for FCB's in this area. See the CONFIG.SYS file information in this section for a description of the "FILES" command. Note that each open file requires its own FCB.

FIELDS OF THE FCB. The FCB is structured as follows:

Name	Size (bytes)	Offset Hex	Offset Decimal
Drive number	1	00H	0
Filename	8	01-08H	1-8
Extension	3	09-0BH	9-11
Current block	2	0CH,0DH	12,13
Record size	2	0EH,0FH	14,15
File size	4	10-13H	16-19
Date of last write	2	14H,15H	20,21
Time of last write	2	16H,17H	22,23
Reserved	8	18-1FH	24-31
Current record	1	20H	32
Relative record	4	21-24H	33-36

Drive Number.
Specifies the disk drive; 1 means drive A: and 2 means drive B: and so on. If the FCB is to be used to create or open a file, this field can be set to 0 to specify the default drive; the Open File system call Function (0FH) sets the field to the number of the default drive.

Filename.
Eight characters, left-aligned and padded (if necessary) with blanks. If you specify a reserved device name (such as PRN), do not put a colon at the end.

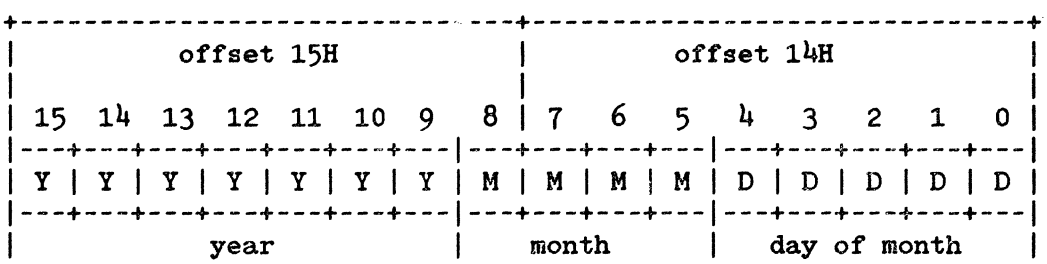
Extension.
Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).

Current Block.
Points to the block (group of 128 records) that contains the current record. This field and the Current Record field (offset 20H) make up the record pointer. This field is set to 0 by the Open File system call.

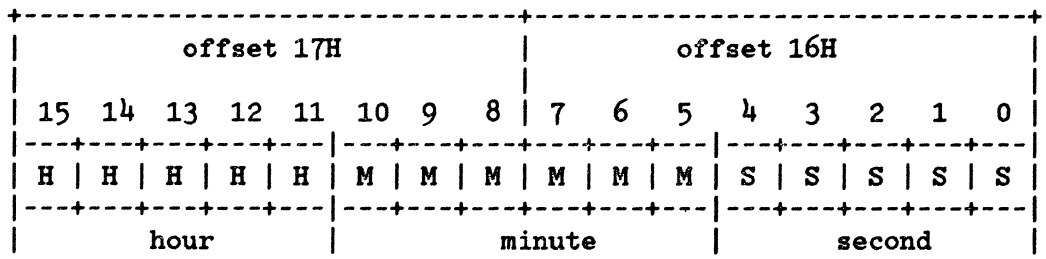
Record Size.
The size of a logical record, in bytes. Set to 128 by the Open File system call. If the record size is not 128 bytes, you must set this field after opening the file.

File Size.
The size of the file, in bytes. The first word of this 4-byte field is the low-order part of the size.

Date of Last Write.
The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:



Time of Last Write.
The time the file was created or last updated. The hour, minutes, and seconds are mapped into two byts as follows:



Reserved. These fields are reserved for use by MS-DOS.

Current Record.
Points to one of the 128 records in the current block. This field and the Current Block field (offset 0CH) make up the record pointer. This field is not initialized by the Open File system call. You must set it before doing a sequential read or write to the file.

Relative Record.
Points to the currently selected record, counting from the beginning of the file (starting with 0). This field is not initialized by the Open File system call. You must set it before doing a random read or write to the file. If the record size is less than 64 bytes, both words of this field are used; if the record size is 64 bytes or more, only the first three bytes are used.

NOTE

If you use the FCB at offset 05CH of the Program Segment Prefix, the last byte of the Relative Record field is the first byte of the unformatted parameter area that starts at offset 80H. This is the default Disk Transfer Address.

Installable Device Drivers

The code for devices additional to those implemented in the BIOS is loaded above the Disc Buffer Cache. The CONFIG.SYS file contains declarations of device driver
files, and these files are loaded from disc and sequentailly linked at system initialization time. For more information on this process and device drivers in general, see the "HP 150 Devices", the "CONFIG.SYS File" and "Operating System Initialization/Booting" discussions later in this section.

PAMCODE.EXE or Resident Portion of COMMAND.COM

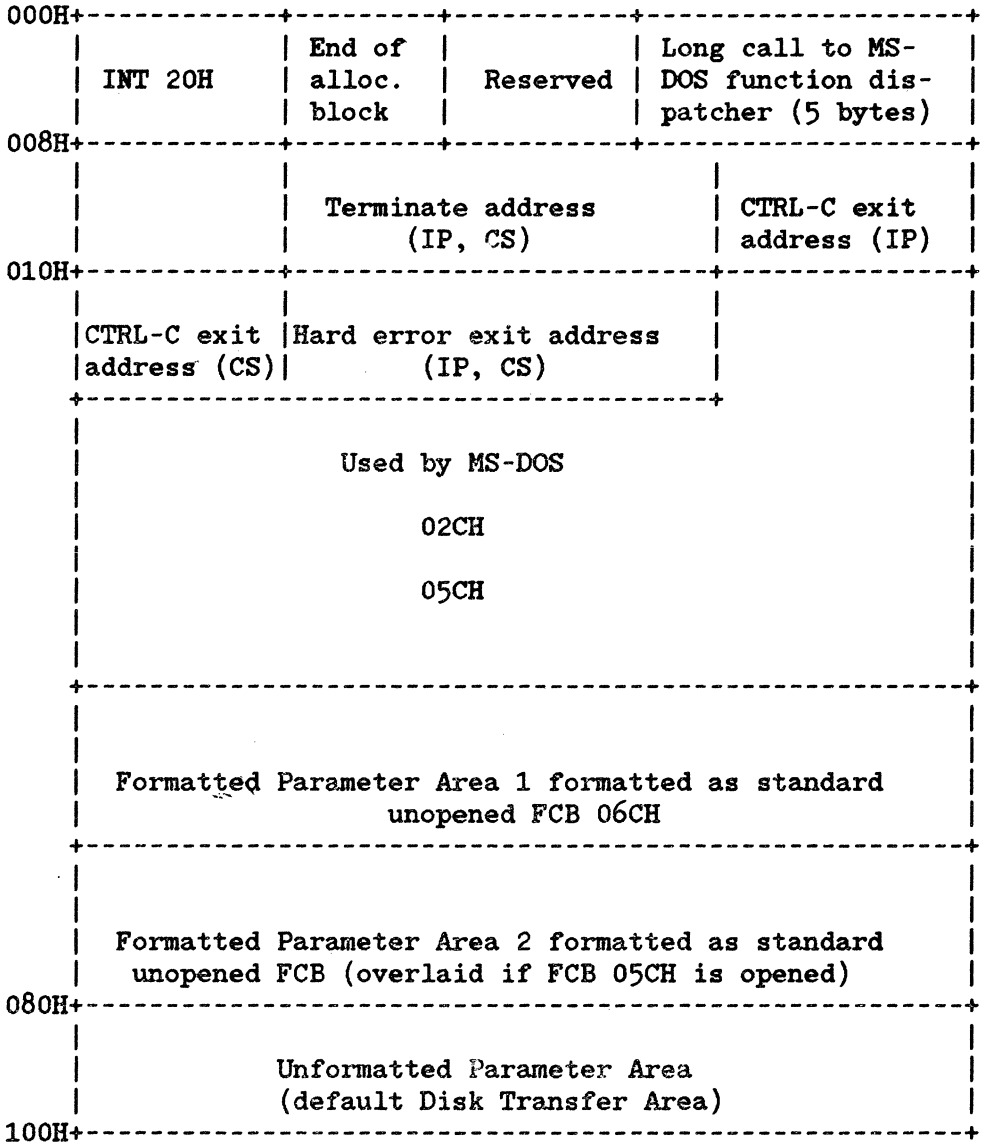
The resident portion of COMMAND.COM contains the code necessary to determine if the transient portion needs to be reloaded (a checksum test), and contains the code to do the reloading of that transient portion. It also contains the interrupt 22H, 23H, and 24H handlers. PAMCODE.EXE is fully resident, that is it does not contain a transient portion in high memory as does COMMAND.COM. As such, if PAMCODE.EXE is declared as the SHELL in the CONFIG.SYS file, quite a deal of system memory will be consumed by this user interface.

Application Program Area (Program Segment)

When an external command is typed, or when a program is executed through P.A.M. or through the EXEC system call, MS-DOS determines the lowest available free memory address to use as the start of the program. This area is called the Program Segment.

The first 256 bytes of the Program Segment are set up by the EXEC system call to use as the Program Segment Prefix (PSP) Control Block.
The program is then loaded following this block. An .EXE file with minalloc and maxalloc both set to zero is
loaded as high as possible.

PROGRAM SEGMENT PREFIX (PSP) CONTROL BLOCK At offset 0 within the program segment, MS-DOS builds the Program Segment Prefix (PSP) control block. The format of this block is as follows:



NOTE

Programs must not alter any part of the Program Segment Prefix control block below offset 05CH.

HOW A PROGRAM TERMINATES

A user (application) program may return to EXEC by one of four methods:

1. A long jump to offset 0 in the Program Segment Prefix (PSP)

2. By issuing an INT 20H with CS:0 pointing to the PSP

3. By issuing an INT 21H with register AH=0 with CS:0 pointing at the PSP, or 4CH and no restrictions on CS

4. By a long call to location 50H in the Program Segment Prefix with AH=0 or Function Request 4CH

NOTE

It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating via any of these methods, except Function Request 4CH. For this reason, using Function Request 4CH is the preferred method.

All four methods result in transferring control to the program that issued the EXEC. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Address, CONTROL-C Exit Address, and Fatal Error Abort Address) addresses are restored from the values saved in the Program Segment Prefix of the terminating program. Control is then given to the terminate address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued; otherwise, COMMAND.COM performs a checksum on the transient part, reloads it if necessary, then issues the system prompt and waits for you to type the next command.

CONDITIONS IN EFFECT WHEN A PROGRAM RECEIVES CONTROL

For All Programs.

The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.

The environment is a series of ASCII strings (totaling less than 32K) in the form:

```
NAME=parameter
```

Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. The environment built by the command processor contains at least a COMSPEC=string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings defined with the MS-DOS SET command.

The environment that is passed is a copy of the invoking process environment. If your application uses a "keep process" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent SET, PATH, or PROMPT commands are issued.

Offset 50H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. By placing the desired function request number in AH, a program can issue a far call to offset 50H to invoke an MS-DOS function, rather than issuing an Interrupt 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.

The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix).

File control blocks at 5CH and 6CH are formatted from the first two parameters typed when the command was entered. If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) will not appear in this area; redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

AL=FFH if the first parameter contained an invalid drive specifier (otherwise AL=00H)

AH=FFH if the second parameter contained an invalid drive specifier (otherwise AH=00H)

Offset 2 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

For .EXE Executable Programs.

DS and ES registers are set to point to the Program Segment Prefix. CS,IP,SS, and SP registers are set to the values passed by MS-LINK.

For .COM Executable Programs.

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.

The Instruction Pointer (IP) is set to 100H.

The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

A word of zeros is placed on top of the stack. This is to allow a user program to exit to COMMAND.COM by doing a RET instruction last. This assumes, however, that the user has maintained his stack and code segments.

Transient Portion of COMMAND.COM

These are the command interpreter, internal commands and batch processing portions of COMMAND.COM. This area may be legally destroyed (overlayed) by an application program, and if it is then it is reloaded from disc at application program termination time.

HP 150 DEVICES

There are two classes of "devices" associated with the HP 150. These are logical devices and physical devices.

Logical (Mappable) Devices

Logical devices can be thought of as generic types of input/output channels. Logical devices are named in a symbolic manner. For example, the device name "PRN" refers to printer, the name "CON" refers to console. The following logical device names are supported by the operating system BIOS Version A.01.02:

```
  * "CON     " - Console Device
    "COM1    " - Primary Communication Device
    "COM2    " - Secondary Communication Device
    "AUX     " - Auxiliary Device
    "PRN     " - Printer Device
  * "CLOCK   " - Clock Device
    "LST     " - List Device
    "PLT     " - Plotter Device
  * "HPIBDEV" - HPIB Device
  * "INT     " - Internal Printer Device
    "LPT1    " - Parallel Printer Device 1 (another name for PRN)
    "LPT2    " - Parallel Printer Device 2 (another name for LST)
    "LPT3    " - Parallel Printer Device 3 (another name for AUX)
"A","B","C"...- Disc Drives
```

Note: * These named devices have actual physical devices associated directly with them and thus are not mappable. All of the other named devices are mappable.

Physical Devices

The physical class of devices on the HP 150 are the hardware input/output entities.

These include the

- Serial Communications Port 1
- Serial Communications Port 2
- Real Time Clock
- Keyboard
- Display Screen
- Integral Printer
- HPIB Port.

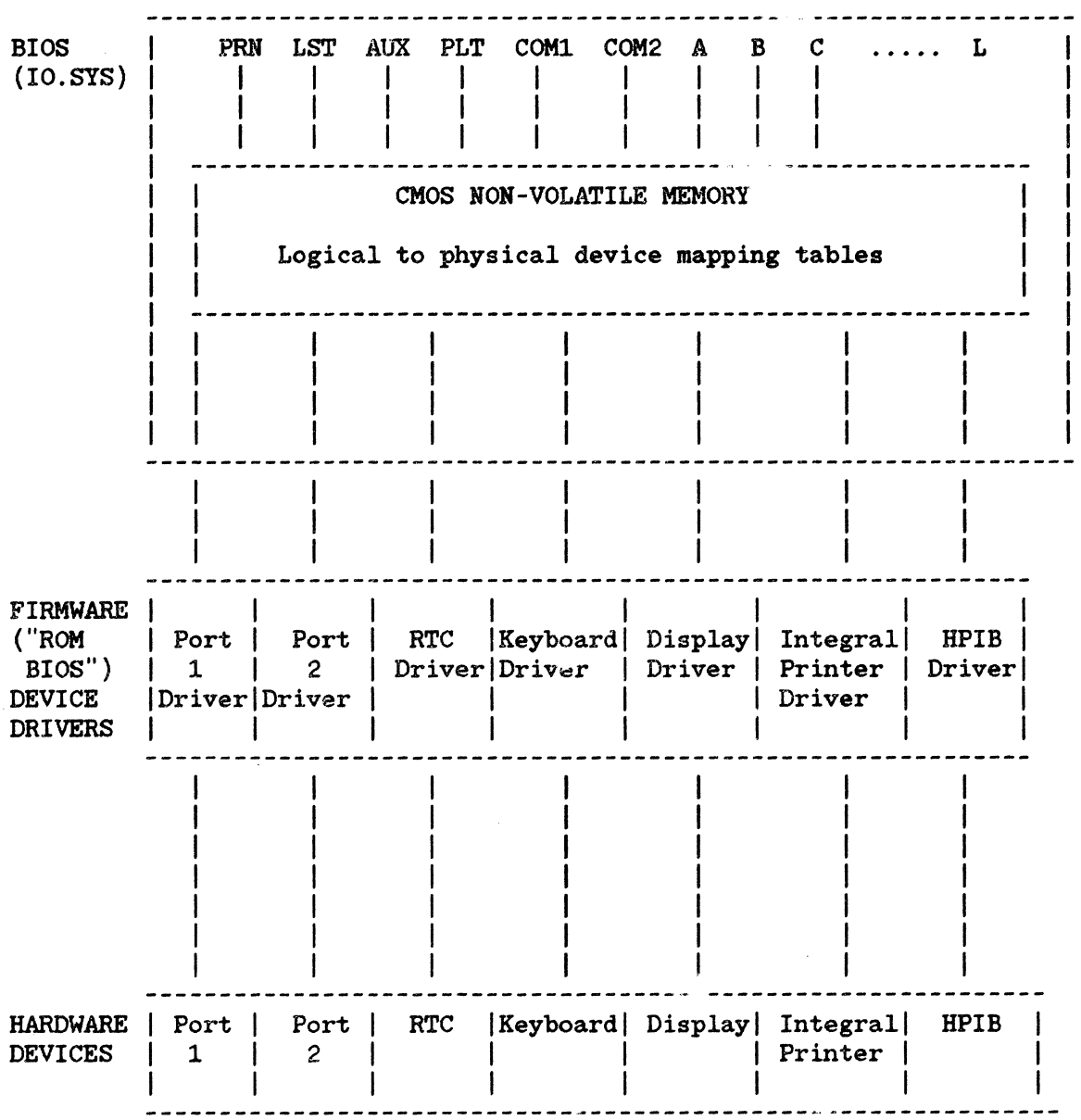
Mapping Logical to Physical Devices

The HP 150 personal computer is uniquely flexible when it comes to the assignment of logical to physical devices. The logical devices may be "mapped"

to physical devices. In this respect, the HP 150 differs from most common personal computers. Some other common personal computers for example have fixed logical to physical device mapping. The serial communications devices COM1 and COM2 are permanently mapped to specific hardware I/O addresses carried on internal board connectors. While this presents a simple system architecture, it also carries with it a rather rigid one.

The Device Configuration Utility

The HP 150 has a piece of non-volatile memory which contains active logical to physical mapping information. An applications program called "DEVCONFG.EXE" allows you to change the mapping stored here.
DEVCONFG is a menu driven application and its usage is described in the HP 150 Owner's Guide. Conceptually, the mapping scheme looks like this:



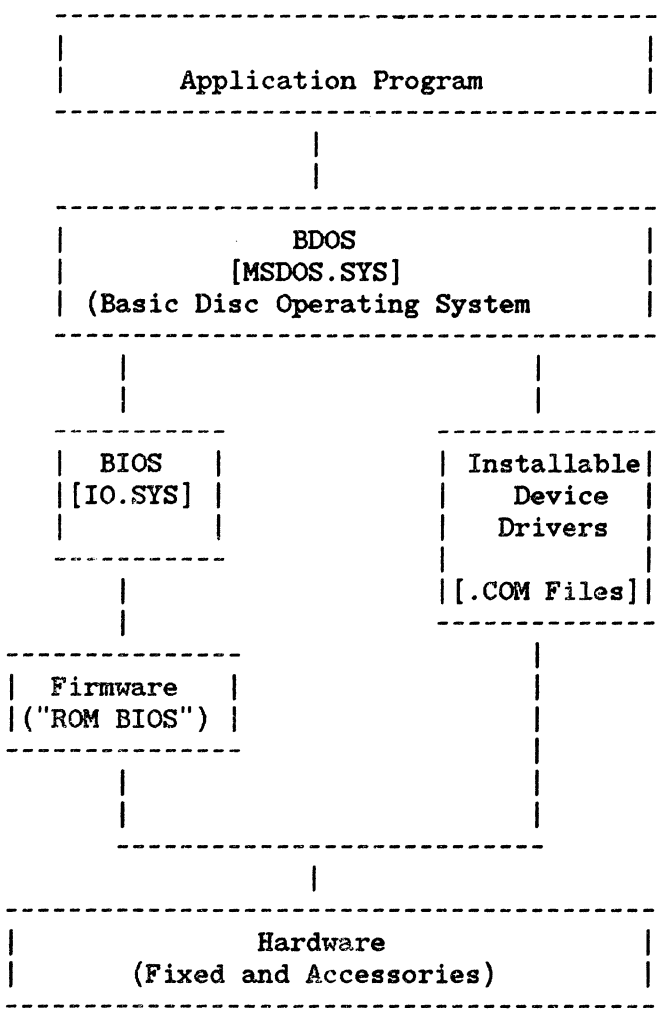
Installable Devices

There is really a third class of devices on the HP 150. These devices are termed "installable devices". They combine the attributes of both logical and physical devices. Installable devices can be created by the system programmer to implement I/O. Installable devices are given names just as the logical devices are however installable devices are implemented outside the BIOS. An example of an installable device might be "NET" -- a network.

Installable devices are really entities unto themselves. They are not mappable (through the CMOS logical to physical device mapping tables) in current HP 150 implementations nor are they recognized by the device configuration utility. Upcoming versions of the operating system will allow installable devices to be mapped into the system device configuration however.

An installable device consists of some executable object code termed a "device driver". The code is stored on the operating system disc as a file with the extension .COM (for example NETWORK.COM). The device driver generally implements all levels of the I/O interface from its BDOS interface down to the hardware itself.

Installable devices can be thought of as extensions to the BIOS. They allow for I/O expansion of an MS-DOS based system. The device architecture is described in the following diagram.



Installable devices share the same calling conventions as the BIOS devices. A discussion of how to implement installable device drivers appears later in this, the system software section.

Character Devices and Block Devices

There are two kinds of devices:

Character devices
Block Cevices

Character devices are designed to perform serial character I/O like CON, AUX, and PRN. These devices are named (i.e., CON, AUX, CLOCK, etc.), and users may open channels (handles or FCBs) to do I/O to them.

Block devices are the "disk drives" on the system. They can perform random I/O in pieces called blocks (usually the physical sector size). These devices are not named as the character devices are, and therefore cannot be opened directly. Instead they are identified via the drive letters (A:, B:, C:, etc.).

Block devices also have units. A single driver may be responsible for one or more disk drives. For example, block device driver ALPHA may be responsible for drives A:,B:,C: and D:. This means that it has four units (0-3) defined and, therefore, takes up four drive letters. If driver ALPHA is the first block driver in the device list, and it defines 4 units (0-3), then they will be A:,B:,C: and D:. If BETA is then the second block driver and defines three units (0-2), then they will be E:,F: and G:, and so on. MS-DOS 2.0 is not limited to 16 block device units, as previous versions were. The theoretical limit is 63 (26 - 1), but it should be noted that after 26 the drive letters are unconventional (such as], \, and .).

NOTE

Character devices cannot define multiple units because they have only one name.

How Application Programs Can Get to Devices

The BDOS (the heart of MS-DOS) serves as an interface for dealing with devices. Many of the MS-DOS System Functions access devices implemented in the BIOS. These System Functions include functions to input and output characters to the CON, AUX, and PRN devices, functions to perform block I/O (reads and writes) to character (CON, AUS, PRN) and block (A,B,C...etc.) devices, and functions to perform block device file directory (create, open, close) operations.

A special I/O control function allows control information to be passed to devices. This may be used for example to perform special CON device functions including the whole AGIOS function library. MS-DOS also allows access to installed devices through many of its system functions.

Application programs for the sake of portability should gain access to devices through the MS-DOS (BDOS) System Function Calls. See "Accessing Devices through MS-DOS" later in this chapter for more information.

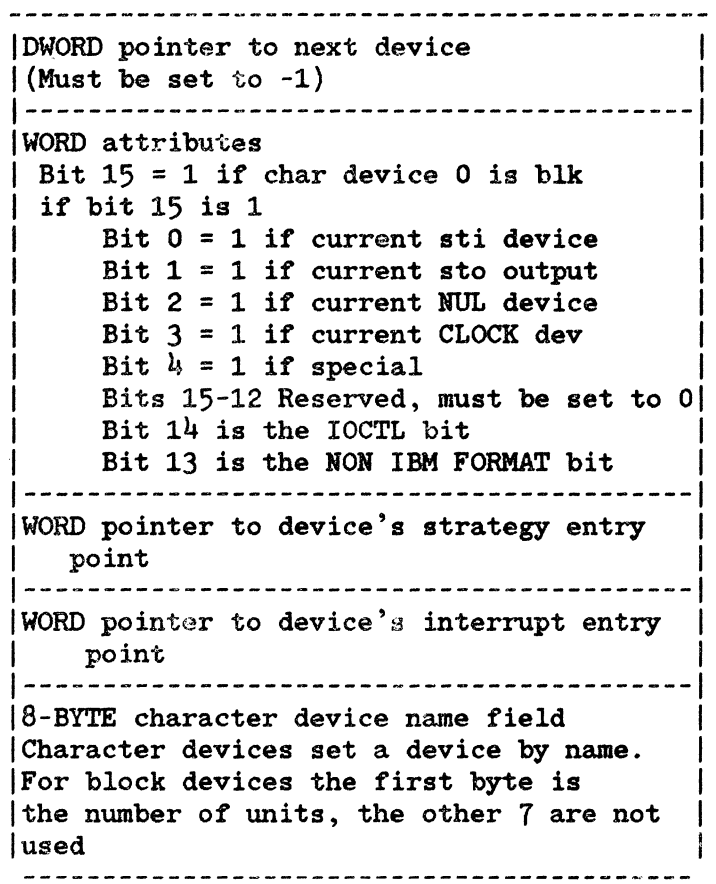
Device Driver Structure

A device driver is a binary file with all of the code in it to manipulate the hardware and provide a consistent interface to MS-DOS. In addition, it has a special header at the beginning that identifies it as a device, defines the strategy and interrupt entry points, and describes various attributes of the device.

NOTE

For device drivers, the file must not use the ORG 100H (like .COM files). Because it does not use the Program Segment Prefix, the device driver is simply loaded; therefore, the file must have an origin of zero (ORG 0 or no ORG statement).

A device header is required at the beginning of a device driver.
A device header looks like this:



Note that the device entry points are words. They must be offsets from the same segment number used to point to this table. For example, if XXX:YYY points to the start of this table, then XXX:strategy and XXX:interrupt are the entry points.

POINTER TO NEXT DEVICE FIELD The pointer to the next device header field is a double word field (offset followed by segment) that is set by MS-DOS to point at the next driver in the system list at the time the device driver is loaded. See the "Device List" discussion which appears later in this chapter. It is important that this field be set to -1 prior to load (when it is on the disk as a file) unless there is more than one device driver in the file. If there is more than one driver in the file, the first word of the double word pointer should be the offset of the next driver's Device Header.

NOTE

If there is more than one device driver in the installable device driver file, the last driver in the file must have the pointer the next Device Header field set to -1.

ATTRIBUTE FIELD

The attribute field is used to tell the system whether this device is a block or character device (bit 15). Most other bits are used to give selected character devices certain special treatment. (Note that these bits mean nothing on a block device). For example, assume that a user has a new device driver that he wants to be the standard input and output. Besides installing the driver, he must tell MS-DOS that he wants his new driver to override the current standard input and standard output (the CON device). This is accomplished by setting the attributes to the desired characteristics, so he would set bits 0 and 1 to 1 (note that they are separate!). Similarly, a new CLOCK device could be installed by setting that attribute. (Refer to "The CLOCK Device", in this chapter for more information.) Although there is a NUL device attribute, the NUL device cannot be reassigned. This attribute exists so that MS-DOS can determine if the NUL device is being used.

The NON IBM FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (Bios Parameter Block) device call. (Refer to "Media Check" and "Build BPB" in the "Calling a Device Driver" section later in this chapter for further information on this call.)

The other bit of interest is the IOCTL bit, which has meaning on character and block devices. This bit tells MS-DOS whether the device can handle control strings (via the IOCTL system call, Function 44H).

If a driver cannot process control strings, it should initially set this bit to 0. This tells MS-DOS to return an error if an attempt is made (via Function 44H) to send or receive control strings to this device. A device which can process control strings should initialize the IOCTL bit to 1. For drivers of this type, MS-DOS will make calls to the IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data to be sent and received by the device for its own use (for example, to set baud rate, stop bits, and form length), instead of passing data over the device channel as does a normal read or write. The interpretation of the passed information is up to the device, but it must not be treated as a normal I/O request.

STRATEGY AND INTERRUPT ROUTINES

These two fields are the pointers to the entry points of the strategy and interrupt routines. The strategy and interrupt routines are the actual device driver code. They are word values so they must be in the same segment as the Device Header.

The 2.0 DOS does not really make use of two entry points (it simply calls strategy, then immediately calls interrupt). This dual entry point scheme is designed to facilitate future multi-tasking versions of MS-DOS. In multi-tasking environments I/O must be asynchronous, to accomplish this the strategy routine will be called to queue (internally) a request and return quickly. It is then the responsibility of the interrupt routine to perform the actual I/O at interrupt time by picking requests off the internal queue (set up by the strategy routine), and process them. When a request is complete, it is flagged as "done" by the interrupt routine. The DOS periodically scans the list of requests looking for ones flagged as done, and "wakes up" the process waiting for the completion of the request.

In order for requests to be queued as above it is no longer sufficient to pass I/O informatin in registers, as was the case in earlier versions, since many requests may be pending at any one time. Therefore the new device interface uses data "packets" to pass request information. A device is called with a pointer to a packet called a Request Header, this packet is linked into a global chain of all pending I/O requests maintained by the DOS. The device then links the packet into its own local chain of requests for this particular device. The device interrupt routine picks requests of the local chain for processing. The DOS scans the global chain looking for completed requests. These packets are composed of two pieces, a static piece which has the same format for all requests (called the static request header), which is followed by information specific to the request. Thus packets have a variable size and format.

At this point it should be emphasized that MS-DOS 2.0 does not implement most of these features, as future versions will. There is no global or local queue. Only one request is pending at any one time, and the DOS waits for this current request to be completed. For 2.0 it is sufficient for the strategy routine to simply store the address of the packet at a fixed location, and for the interrupt routine to then process this packet by doing the request and returning.

Remember: The DOS just calls the strategy routine and then immediately calls the interrupt routine, it is assumed that the request is completed when the interrupt routine returns.

NAME FIELD

This is an 8-byte field that contains the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional, because MS-DOS will fill in this location with the value returned by the driver's INIT code. Refer to "Installation of Device Drivers" in this chapter for more information.

Device List

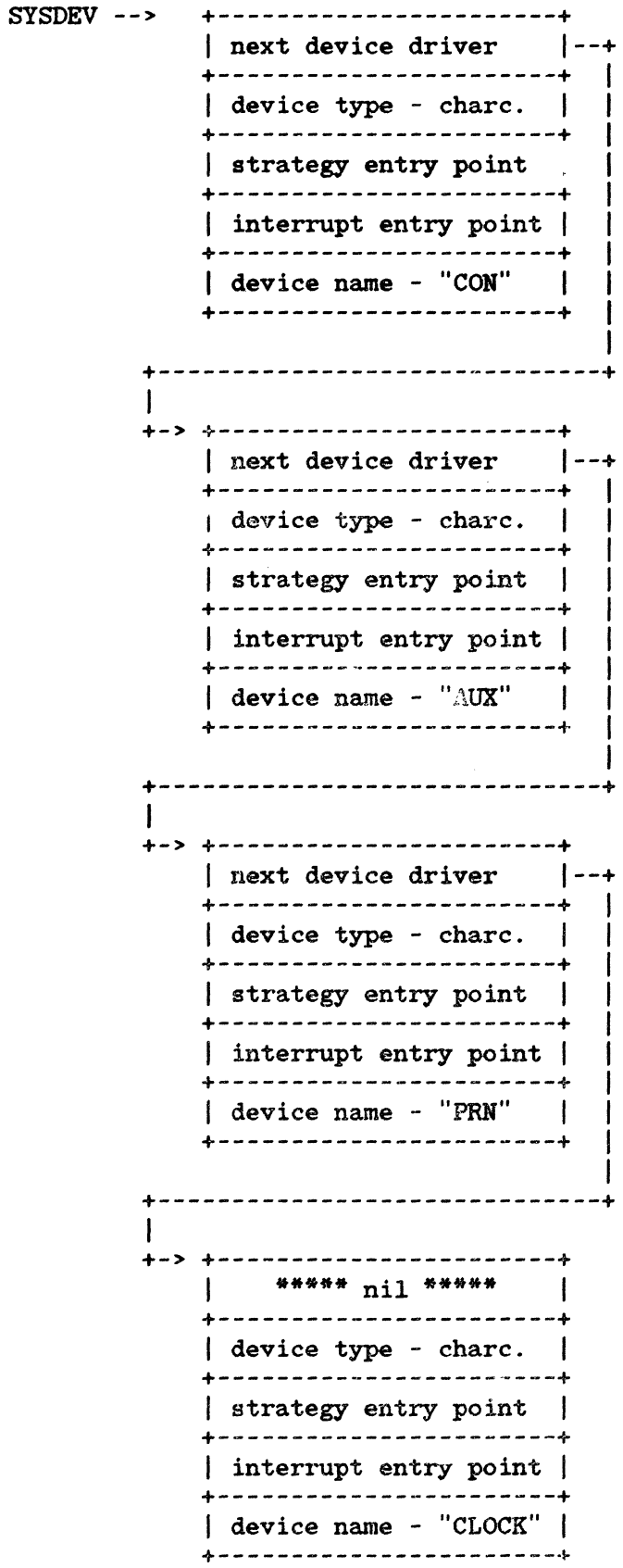
A list of all devices accessible by the operating system is maintained in RAM memory on the HP 150. The list includes each device supported by the BIOS together with any installable device drivers loaded from disc during initialization of the operating system.

During operating system initialization, the SYSINIT routine provided by Microsoft the BIOS reads a file called CONFIG.SYS located on the operating system disc. If any installable device drivers are to be installed, the CONFIG.SYS file should contain an entry such as

```
DEVICE = NETWORK.SYS
```

for each installable device. SYSINIT will load the device driver from disc (the file "NET.COM" in the above example) and link it into the list of existing device drivers. This device list contains both the boot (default BIOS) drivers and any installable device drivers found referenced in the CONFIG.SYS file. The system always processes the installable device drivers first and links them in ahead of the default ones, thus allowing the user to override default devices if he so chooses.

The format of the entries in the system's device list is the same as that of the device header required at the beginning of an installable device driver file. The details of this structure have been described, but in general it consists of the device specific information MSDOS needs to utilize each device. Such entries as the device type, and pointers to the device driver's strategy and interrupt routines are found in the system's device list structure. MSDOS requires that there be at least four devices defined by this list at boot time. They must be both the first four entries in this list and they must implement the CON, AUX, PRN, and CLOCK devices. The minimum system device list would then appear as follows:


SYSDEV -->
next device driver
device type - charc.
strategy entry point
interrupt entry point
device name - "CON"
next device driver
device type - charc.
strategy entry point
interrupt entry point
device name - "AUX"
next device driver
device type - charc.
strategy entry point
interrupt entry point
device name - "PRN"
***** nil *****
device type - charc.
strategy entry point
interrupt entry point
device name - "CLOCK"

An installable device driver with the same name as one of these BIOS resident devices will be linked in at that position, thus preempting the default device of the same name. As such, BIOS devices may be replaced by installable devices of the same name. Additional devices of either type (block or character) will be added to the system (whether they reside in BIOS or on disc) by updating the "next device driver" field of the last entry in this list to the point to the additional drivers. The SYSINIT module of the BIOS is responsible for bringing in the installable device drivers from the disc and inserting them in this list as pictured above. Once all of the installable drivers have been linked in, MS-DOS scans the preset list of default or BIOS resident device drivers, and links them into this list. The logical device names for the block devices (the labels A, B, C, etcetera) are determined by the position of the block device in the device list, and by how many units each device driver supports. The first unit of the first block device driver in the device list is assigned the label A, the second B, and so on.

How to Create a Device Driver

In order to create a device driver that MS-DOS can install, you must write a binary file with a Device Header at the beginning of the file. Note that for device drivers, the code should not be originated at 100H, but rather at 0. The link field (pointer to next Device Header) should be -1, unless there is more than one device driver in the file. The attribute field and entry points must be set correctly.

If it is a character device, the name field should be filled in with the name of that character device. The name can be any legal 8-character filename.

MS-DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on a new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

NOTE

Because MS-DOS can install the driver anywhere in memory, care must be taken in any far memory references. You should not expect that your driver will always be loaded in the same place every time.

HOW MS-DOS CALLS A DEVICE DRIVER

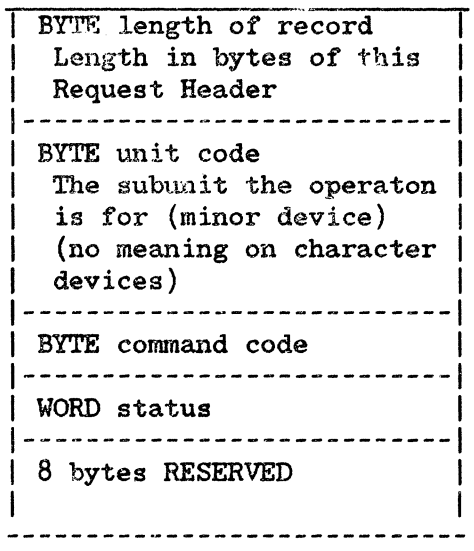
Request Header

When MS-DOS receives a reference to a device it scans the device list to find the first match of the referenced device. From this list it pulls the pointer to the device's strategy routine and calls the code with a device request header. The strategy routine simply saves a pointer to the segment and offset of the start of the device request header, and returns to the DOS. The DOS then immediatly calls the device's interrupt routine, where the request header is examined and acted upon as requested. It is the responsibility of the device's interrupt routine to examine the command code in the request header, transfer control of the routine to process that command, set the status word in the request header appropriatly, fill in any command specific parameters, and return to the DOS.

A pointer to the Request Header is passed to the strategy entry point in ES:BX. is made up of a piece of fixed-length data (the Static Request Header) followed by data pertinent to the operation being performed. Note that is is the device driver's responsibility to preserve the machine state (for example, save all registers on entry and restore them on exit). There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver should set up its own stack.

The following figure illustrates the static portion of a Request Header.

STATIC REQUEST HEADER ->



UNIT CODE The unit code field identifies which unit of a block device in your

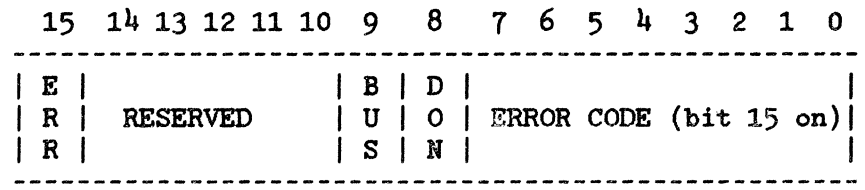
device driver the request is for. For example, if your device driver has 3 units defined, then the possible values of the unit code field would be 0, 1, and 2.

COMMAND CODE FIELD The command code field in the Static Request Header specifies the function to be performed and can have the following values:

Command Code	Function
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB (Block only, NOP for character)
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS (Char devs only)
7	INPUT FLUSH (Char devs only)
8	OUTPUT (write)
9	OUTPUT (write) with verify
10	OUTPUT STATUS (Char devs only)
11	OUTPUT FLUSH (Char devs only)
12	IOCTL OUTPUT (Only called if device has IOCTL)

These functions are described in detail later in the section.

STATUS WORD The following figure illustrates the status word in the Request Header.



The status word is zero on entry and is set by the driver interrupt routine on return.

Bit 8 is the one bit. When set, it means the operation is complete. For MS-DOS 2.0, the driver sets it to 1 when it exits.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error. The errors are:

0 Write protect violation
1 Unknown Unit
2 Drive not ready
3 Unknown command
4 CRC error
5 Bad drive request structure length
6 Seek error
7 Unknown media
8 Sector not found
9 Printer out of paper
A Write fault
B Read fault
C General failure

Bit 9 is the busy bit, which is set only by status calls.

For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request, and a write request (if made) would start immediately.

For input on character devices with a buffer: If bit 9 is 1 on return, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the device buffer and a read would return quickly. It also indicates that something has been typed. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy=0 so that MS-DOS will not continuously wait for something to get into a buffer that does not exist.

One of the functions defined for each device is INIT. This routine is called only once when the device is installed. The INIT routine returns a location (DS:DX), which is a pointer to the first free byte of memory after the device driver (similar to "Keep Process"). This pointer method can be used to delete initialization code that is only needed once, saving on space.

Block devices are installed the same way and also return a first free byte pointer as described above. Additional information is also returned:

The number of units is returned. This determines logical device names. If the current maximum logical device letter is F at the time of the install call, and the INIT routine returns 4 as the number of units, then they will have logical names G, H, I, and J. This mapping is determined by the position of the driver in the device list, and by the number of units on the device (stored in the first byte of the device name field).

A pointer to a BPB (BIOS Parameter Block) pointer array is also returned. There is one table for each unit defined. These blocks will be used to build an internal DOS data structure for each of the units. The pointer passed to the DOS from the driver points to an array of n word pointers to BPBs, where n is the number of units defined. In this way, if all units are the same, all of the pointers can point to the same BPB, saving space. Note that this array must be protected (below the free pointer set by the return) since an internal DOS structure will be built starting at the byte

pointed to by the free pointer. The sector size defined must be less than or equal to the maximum sector size defined at default BIOS INIT time. If it isn't, the install will fail.

The last thing that INIT of a block device must pass back is the media descriptor byte. This byte means nothing to MS-DOS, but is passed to devices so that they know what parameters MS-DOS is currently using for a particular drive unit.

Block devices may take several approaches; they may be dumb or smart. A dumb device defines a unit (and therefore an internal DOS structure) for each possible media drive combination. For example, unit = drive 0 single side, unit 1 = drive 0 double side. For this approach, media descriptor bytes do not mean anything. A smart device allows multiple media per unit. In this case, the BPB table returned at INIT must define space large enough to accommodate the largest possible media descriptor byte to pass information about what media is currently in a unit.

Device Driver Functions and Parameters

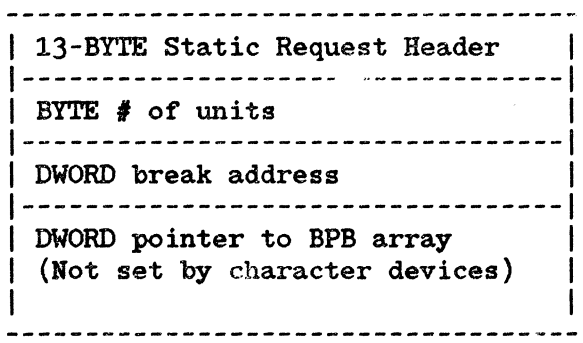
All strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the Request Header tells the driver which function to perform.

NOTE

All DWORD pointers are stored offset first, then segment.

INIT
Command code = 0

INIT - ES:BX ->



The number of units, break address, and BPB pointer are set by the driver. On entry, the DWORD that is to be set to the BPB array (on block devices) points to the character after the '=' on the line in CONFIG.SYS that loaded this device. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

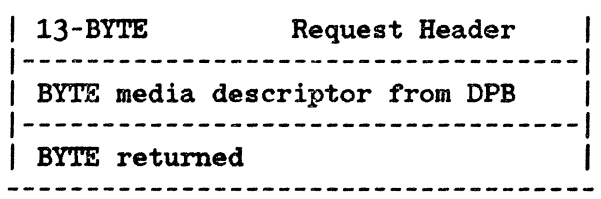
NOTE

If there are multiple device drivers in a single device driver file, the ending address returned by the last INIT called will be the one MS-DOS uses. It is recommended that all of the device drivers in a single device driver file return the same ending address.

MEDIA CHECK ***Command Code = 1***

MS-DOS calls MEDIA CHECK first for a drive unit. MS-DOS passes its current media descriptor byte (refer to the section "Media Descriptor Byte" later in this chapter).

MEDIA CHECK - ES:BX ->



In addition to setting the status word, the driver must set the return byte to one of the following:

Media Not Changed - current DPB and media byte are OK.

Media Changed - Current DPB and media are wrong. MS-DOS invalidates any buffers for this unit and calls the device driver to build the BPB with media byte and buffer.

Not Sure If there are dirty buffers (buffers with changed data, not yet written to disk) for this unit, MS-DOS assumes the DPB and media byte are OK (media not changed). If nothing is dirty, MS-DOS assumes the media has changed. It invalidates any buffers for the unit, and calls the device driver to build the BPB with media byte and buffer.

Error - If an error occurs, MS-DOS sets the error code accordingly.

-1 Media has been changed
 0 Don't know if media has been changed
 1 Media has not been changed

If the driver can return -1 or 1 (by having a door-lock or other interlock mechanism) MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT for each directory access.

BUILD BPP (BIOS PARAMETER BLOCK)
Command code = 2

MS-DOS will call BUILD BPB under the following conditions:

If Media Changed is returned

If Not Sure is returned, and there are no dirty buffers.

The BUILD BPB call also gets a pointer to a one-sector buffer. What this buffer contains is determined by the NON IBM FORMAT bit in the attribute field. If the bit is zero (device is IBM format-compatible), then the buffer contains the first sector of the first File Allocation Table (FAT). The FAT IB byte is the first byte of this buffer.

NOTE

The BPB must be the same, as far as location of the FAT is concerned, for all possible media because this first FAT sector must be read before the actual BPB is returned. If the NON IBM FORMAT bit is set, then the pointer points to one sector of scratch space (which may be used for anything).

BUILD BPB - ES:BX ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address (Points to one sector worth of scratch space or first sector of FAT depending on the value of the NON IBM FORMAT bit)
DWORD pointer to BPB

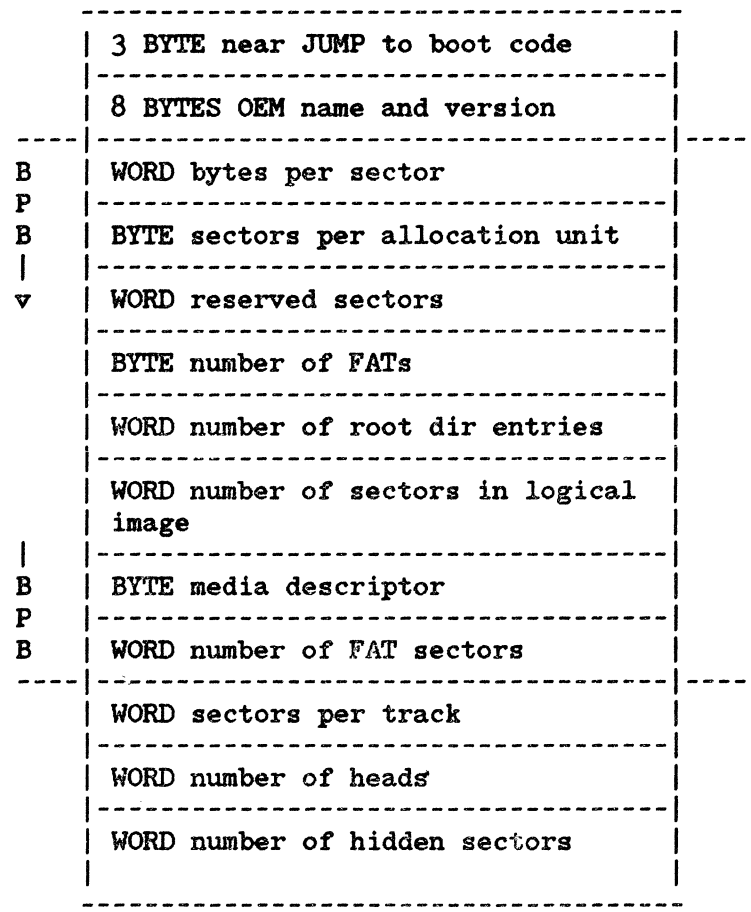
If the NON IBM FORMAT bit of the device is set, then the DWORD transfer address points to a one sector buffer, which can be used for any purpose. If the NON IBM FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

If IBM compatible format is used (NON IBM FORMAT BIT = 0), then the first sector of the first FAT must be located at the same sector on all possible media. This is because the FAT sector will be read BEFORE the media is actually determined. Use this mode if all you want is to read the FAT ID byte.

In addition to setting status word, the driver must set the Pointer to the BPB on return.

In order to allow for many different OEMs to read each other's disks, the following standard is suggested. The information relating to the BPB for a particular piece of media is kept in the boot sector for the media. In particular, the format of the boot sector is:

BUILD BPP - ES:BX ->



The three words at the end (sectors per track, number of heads, and number of hidden sectors) are optional. They are intended to help the BIOS understand the media. Sectors per track may be redundant (could be calculated from total size of the disk). Number of heads is useful for supporting different multi-head drives which have the same storage capacity, but different numbers of surfaces. Number of hidden sectors may be used to support drive-partitioning schemes.

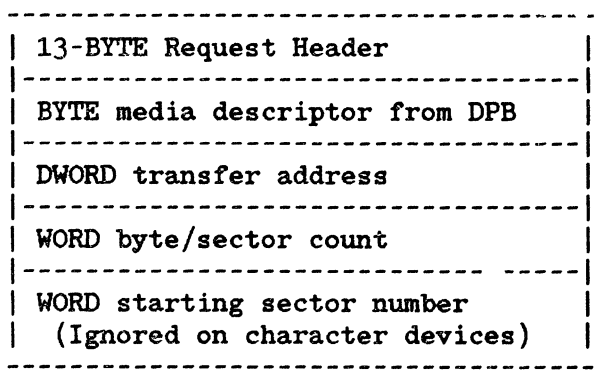
Media Descriptor Byte

The last two digits of the FAT ID byte are called the media descriptor byte. Currently, the media descriptor byte has been defined for a few media types, including 5-1/4" and 8" standard disks.

Although these media bytes map directly to FAT ID bytes (which are constrained to the 8 values F8H-FFH), media bytes can, in general, be any value in the range 00H-FFH.

READ OR WRITE *Command codes = 3,4,8,9, and 12*

READ or WRITE - ES:BX (Including IOCTL) ->

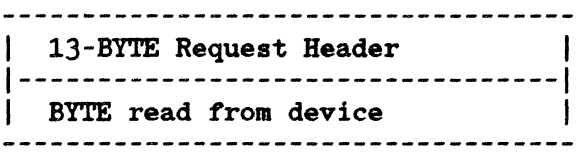


In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must correctly set the return sector (byte) count to the actual number of bytes transferred.

The Following Applies to Block Device Drivers:
Under certain circumstances the BIOS may be asked to perform a write operation of 64K bytes, which seems to be a "wrap around" of the transfer address in the BIOS I/O packet. This request arises due to an optimization added to the write code in MS-DOS. It will only manifest on user writes that are within a sector size of 64K bytes on files "growing" past the current EOF. It is allowable for the BIOS to ignore the balance of the write that "wraps around" if it so chooses. For example, a write of 10000H bytes worth of sectors with a transfer address of XXX:1 could ignore the last two bytes. A user program can never request an I/O of more than FFFFH bytes and cannot wrap around (even to 0) in the transfer segment. Therefore, in this case, the last two bytes can be ignored.

NON DESTRUCTIVE READ NO WAIT
Command code = 5

NON DESTRUCTIVE READ NO WAIT - ES:BX ->

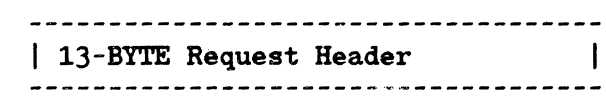


If the character device returns bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer (hence the term "Non Destructive Read"). Basically, this call allows MS-DOS to look ahead one input character.

STATUS

Command codes = 6 and 10

STATUS - ES:BX ->



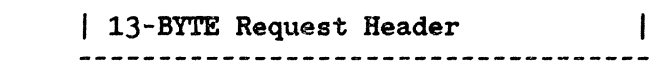
All the driver must do is set the status word and the busy bit as follows:

For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request and a write request (if made) would start immediately.

For input on character devices with a buffer: A return of 1 means, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the devices buffer and read would return quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0 so that the DOS will not hang waiting for something to get into a buffer which doesn't exist.

FLUSH ***Command codes = 7 and 11***

FLUSH - ES:BX ->



The FLUSH call tells the driver to flush (terminate) all pending requests. This call is used to flush the input queue on character devices.

HP 150 INSTALLABLE DEVICE DRIVER EXAMPLE

This is an example of an MSDOS installable device driver for the HP 150. As an MSDOS installable device driver, this example does nothing more than support the 12 MSDOS command codes in name only, and does not attempt to implement their functions. The intention of the example is to outline the structure of an installable device driver. It is then a matter for the programmer to "fill in the blanks" to implement a functional driver. All device functions exist in name only, returning done status immediatly.

```
;*****************************************************************************
;
;                    Driver's MSDOS Command Code Routines
;
;
; Driver Routine's Label          Function of the particular routine
; ------------------------------------------------------------------------
; MSDOS_COMMAND_CODES:            Supported MDSOS Command Codes
;    MSDOS_INITIALIZE             0 - initializes the option into the system
;    MSDOS_MEDIA_CHECK            1 - performs media check on block devices
;    MSDOS_BUILD_BPB              2 - builds the device's BIOS Parameter Block
;    MSDOS_IOCTL_INPUT            3 - performs an I/O control read from device
;    MSDOS_INPUT                  4 - performs a normal destructive read
;    MSDOS_NON_DSTRV_INPUT        5 - performs a non-destructive read, no wait
;    MSDOS_INPUT_STATUS           6 - returns current input status of device
;    MSDOS_INPUT_FLUSH            7 - flushes the device's input buffers
;    MSDOS_OUTPUT                 8 - performs a normal output to the device
;    MSDOS_OUTPUT_WITH_VERIFY     9 - performs output with verify to the device
;    MSDOS_OUTPUT_STATUS         10 - returns current output status of device
;    MSDOS_OUTPUT_FLUSH          11 - flushes the device's output buffers
;    MSDOS_IOCTL_OUTPUT          12 - performs IO control output to the device
;
;
; NOTE:   All command code routines listed above exist in name only and simply
;         return status equal to done when called.
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                    Start of the Installable Device Driver Routine
;
;
;   This next section starts the code segment for the rest of this module.
;
CODE    SEGMENT PUBLIC 'CODE'
DRIVER  PROC    FAR
        ASSUME  CS:CODE,ES:CODE,DS:CODE
        ORG     0
BEGIN:
START   EQU     $
;
;*****************************************************************************

;*****************************************************************************
;
;                    MSDOS Installable Device Driver Header Format
;
;
;    The following is the MSDOS installable device driver's device header.
; This header must be at the beginning of each installable device driver. It
; is used by the SYSINIT module provided by Microsoft to link this installable
; device driver into the system's device list. The format of this device header
; is as follows:
;
;      15   14   13   12   11   10   9   8   7   6   5   4   3   2   1   0
;    +-------------------------------------------------------------------+
;    | offset of the NEXT DEVICE DRIVER in the device list (-1 if last entry)|
;    +-------------------------------------------------------------------+
;    | segment of the NEXT DEVICE DRIVER in the device list(-1 if last entry)|
;    +-------------------------------------------------------------------+
;    | ATTRIBUTES word describing device specific characteristics        |
;    +-------------------------------------------------------------------+
;    | offset to the device driver's STRATEGY routine                    |
;    +-------------------------------------------------------------------+
;    | offset to the device driver's INTERRUPT routine                   |
;    +-------------------------------------------------------------------+
;    | 1st 2 bytes of a charac. dev. NAME, or # of UNITS for a block device  |
;    +-------------------------------------------------------------------+
;    | 2nd 2 bytes of a character device NAME, or nothing for a block device |
;    +-------------------------------------------------------------------+
;    | 3rd 2 bytes of a chacacter device NAME, or nothing for a block device |
;    +-------------------------------------------------------------------+
;    | 4th 2 bytes of a character device NAME, or nothing for a block device |
;    +-------------------------------------------------------------------+
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                    Driver's Installable Device Driver Header
;
;
;   The device implemented by the driver is a simple character device
; with an MSDOS strategy routine at DRIVER_STRATEGY and an MSDOS interrupt
; routine at DRIVER_INTERRUPT.  The name of this device is "DRIVER".  The
; installable device driver header for this device is as follows:
;
NEXT_DEVICE     DD      -1                      ;link to next dev.=-1 end of list
ATTRIBUTES      DW      08000H                  ;8000H means a character device
STRATEGY        DW      DRIVER_STRATEGY         ;driver's STRATEGY routine entry
INTERRUPT       DW      DRIVER_INTERRUPT        ;driver's INTERRUPT routine entry
DEVICE_NAME     DB      "DRIVER  "              ;driver's name is "DRIVER"
;
;*****************************************************************************

;*****************************************************************************
;
;                Driver's MSDOS Command Code Routines Dispatch Table
;
;
;    The following section contains a dispatch table that describes the entry
; points for the routines that implement the various 12 MSDOS command codes for
; this example device.  This table is indexed into based on the MSDOS command
; code in the request header, and a jump is made to the address of the routine
; listed in this table.  The dispatch table of MSDOS command codes for this
; example driver is as follows:
;
MSDOS_COMMAND_CODES:                ;Support MSDOS Command Codes (dispatch table)
   DW  MSDOS_INITIALIZE             ; 0 - initializes the driver into the system
   DW  MSDOS_MEDIA_CHECK            ; 1 - performs media check on block devices
   DW  MSDOS_BUILD_BPB              ; 2 - builds the device's BIOS Parameter Block
   DW  MSDOS_IOCTL_INPUT            ; 3 - performs an I/O control read from device
   DW  MSDOS_INPUT                  ; 4 - performs a normal destructive read
   DW  MSDOS_NON_DSTRV_INPUT        ; 5 - performs a non-destructive read, no wait
   DW  MSDOS_INPUT_STATUS           ; 6 - returns current input status of device
   DW  MSDOS_INPUT_FLUSH            ; 7 - flushes the device's input buffers
   DW  MSDOS_OUTPUT                 ; 8 - performs a normal output to the device
   DW  MSDOS_OUTPUT_WITH_VERIFY     ; 9 - performs output with verify to the device
   DW  MSDOS_OUTPUT_STATUS          ;10 - returns current output status of device
   DW  MSDOS_OUTPUT_FLUSH           ;11 - flushes the device's output buffers
   DW  MSDOS_IOCTL_OUTPUT           ;12 - performs I/O control output to the device
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                          MSDOS Request Header Format
;
;
;    This next section contains a template for the MSDOS request header. This
; template is used to extract and update the necessary request header entries.
; The MSDOS request header's format is as shown below:
;
;
;      7     6     5     4     3     2     1     0
;    +-----------------------------------------------+
;    | LENGTH of the request HEADER in bytes         |
;    +-----------------------------------------------+
;    | UNIT that this call is intended for           |
;    +-----------------------------------------------+
;    | COMMAND CODE or function number               |
;    +-----------------------------------------------+
;    | driver's RETURN STATUS, first of two          |
;    +-----------------------------------------------+
;    | driver's RETURN STATUS, second of two         |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | **********      RESERVED      *********       |
;    +-----------------------------------------------+
;    | ******  Command Specific Data  ******         |
;    .                     .                         .
;    .                     .                         .
;    .                     .                         .
;    | ******  Command Specific Data  ******         |
;    +-----------------------------------------------+
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                    Driver's MSDOS Request Header Structure
;
;
;   The REQUEST_HEADER structure that this module will use to access the MSDOS
; request header packet is defined as follows:
;
REQUEST_HEADER      STRUC                   ;request header template structure

  HEADER_LENGTH     DB  ?                   ;length of the request header
  UNIT_NUMBER       DB  ?                   ;unit number for this request
  COMMAND_CODE      DB  ?                   ;request header's command code
  RETURN_STATUS     DW  ?                   ;driver's return status word
  RESERVED          DB  8   DUP (?)         ;******* RESERVED *******
  MEDIA_DESCRIPTOR  DB  ?                   ;media descriptor byte
  TRANSFER_ADDRESS  DD  ?                   ;transfer address
  COUNT             DW  ?                   ;byte/sector count value
  STARTING_SECTOR   DW  ?                   ;starting sector value

REQUEST_HEADER      ENDS                    ;end of request header template
;
;*****************************************************************************

;*****************************************************************************
;
;                           Driver's Local Variables
;
;
HEADER_POINTER      DD    ?                 ;pointer to the request header
;
;*****************************************************************************

;*****************************************************************************
;
;                           DRIVER's Local Constants
;
;
MAX_COMMAND_CODE    EQU   012               ;MSDOS command code number maximum
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                    Driver's MSDOS Device Driver Strategy Routine
;
;
;    This next section contains this driver's strategy routine.  This routine
; is entered each time a request is made for this device.
;
;    All requests for this device
; first call its strategy routine and then its interrupt routine
; The strategy routine is passed a pointer (in ES:BX) to the request header,
; which it saves in a local variable called HEADER_POINTER and then it returns
; to the BDOS.  This example device driver's strategy routine is defined as
; follows:
;
;                INPUTS  -  ES:BX points to the request header
;
;                OUTPUTS -  HEADER_POINTER points to the active request header
;
;*****************************************************************************
;
STRATEGY_PROCEDURE PROC FAR                     ;beginning of strategy procedure

DRIVER_STRATEGY:                                ;strategy routine entry point
      MOV  WORD PTR CS:[HEADER_POINTER],BX      ;save offset of request header
      MOV  WORD PTR CS:[HEADER_POINTER+2],ES    ;save segment of request header
      RET                                       ;return to the calling routine

STRATEGY_PROCEDURE ENDP                         ;end of the strategy procedure
;
;*****************************************************************************

;*****************************************************************************
;
;                    Driver's MSDOS Device Driver Interrupt Routine
;
;
;    This next section contains this driver's MSDOS interrupt routine.  This
; is the entry point that the BDOS will call immediately after
; it has called this driver's strategy routine.  It is this routine's
; responsibility to transfer control to the appropriate routine within this
; module based on the command code found in the current request header.  It
; uses this command code as an index into its MSDOS command codes dispatch
; table, to extract the address of the routine that can service the incoming
; request.
;
;                INPUTS  -  HEADER_POINTER points to the request header
;
;                OUTPUTS -  REQUEST_HEADER [RETURN_STATUS] set appropriately
;                        -  command code specific data where specified
;
```

```
;*****************************************************************************
;
;
DRIVER_INTERRUPT:                       ;device driver interrupt routine
     PUSH  AX                           ;save the caller's AX register
     PUSH  BX                           ;save the caller's BX register
     PUSH  CX                           ;save the caller's CX register
     PUSH  DX                           ;save the caller's DX register
     PUSH  DS                           ;save the caller's DS register
     PUSH  ES                           ;save the caller's ES register
     PUSH  DI                           ;save the caller's DI register
     PUSH  SI                           ;save the caller's SI register
     PUSH  BP                           ;save the caller's BP register
;
;    These sections of code prepare the system's registers with the values
; found in the request header before transferring control to the appropriate
; routine.  The registers are prepared as follows:
;
;            AX = AH-->MEDIA DESCRIPTOR/AL-->UNIT NUMBER
;            BX = OFFSET within ES segment to the request header
;            CX = COUNT (bytes or sectors) of the data to be read or written
;            DX = STARTING SECTOR to be read or written
;            DI = TRANSFER ADDRESS of user's buffer or data
;            SI = address of the start of the servicing routine
;            DS = data segment address of this driver (same as CS)
;            ES = segment address of the request header
;
;
     LDS   BX,CS:[HEADER_POINTER]       ;point DS:BX to the request header
     MOV   AL,[BX.UNIT_NUMBER]          ;load AL with the UNIT NUMBER
     MOV   AH,[BX.MEDIA_DESCRIPTOR]     ;load AH with the MEDIA DESCRIPTOR
     MOV   CX,[BX.COUNT]                ;load CX with the COUNT value
     MOV   DX,[BX.STARTING_SECTOR]      ;load DX with the STARTING SECTOR
     XCHG  DI,AX                        ;save AX in DI temporarily
;
;    Extract the command code from the request header and branch to the
; appropriate routine within this module that can service it.
;
     MOV   AL,[BX.COMMAND_CODE]         ;load AX with the COMMAND CODE
     XOR   AH,AH                        ;clear off the high byte of AX
     CMP   AL,MAX_COMMAND_CODE          ;is this a legal command code?
     JG    COMMAND_CODE_ERROR           ;if no. then error out here
     SHL   AL,1                         ;convert byte to word pointer
     MOV   SI,OFFSET MSDOS_COMMAND_CODES ;point SI to MSDOS dispatch table
     ADD   SI,AX                        ;index SI to the desired routine
     XCHG  AX,DI                        ;restore AX from DI temp. storage
     LES   DI,[BX.TRANSFER_ADDRESS]     ;point DI to transfer addr. offset
     PUSH  CS                           ;place code segment on the stack..
     POP   DS                           ;then point DS to the code segment
     JMP   WORD PTR [SI]                ;jump to the appropriate routine
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;               Driver's MSDOS Command Code General Purpose Routines
;
;
;     These are a set of general purpose routines that are used by all
;     of the MSDOS command code routines.
;
COMMAND_CODE_ERROR:
     MOV       AL,3                          ;AL=3 means bad command code error
     MOV       AH,10000001B                  ;AH=error and done bits set
     JMP       EXIT_DRIVER                   ;exit driver through EXIT_DRIVER

EXIT:
     MOV       AH,00000001B                  ;AH=no errors occured this time
                                             ;drop through to exit the driver
;
;*****************************************************************************

;*****************************************************************************
;
;                       Driver's MSDOS Command Code Exit Routine
;
;
; This procedure is used to generate a long return to the caller.
; The value passed to this routine in AX is placed in the
; RETURN_STATUS word entry of the pending request header.  Then this routine
; restores the stack back to the state it was when this driver was originally
; called, and then returns to the caller.
;
;
;               INPUTS  - AX contains the RETURN_STATUS value for the
;                         pending I/O request.
;
;
EXIT_PROC  PROC  FAR

EXIT_DRIVER:
     LDS       BX,CS:[HEADER_POINTER]        ;point BX to the request header
     MOV       DS:[BX.RETURN_STATUS],AX      ;set the STATUS word accordingly
     POP       BP                            ;restore the caller's BP register
     POP       SI                            ;restore the caller's SI register
     POP       DI                            ;restore the caller's DI register
     POP       ES                            ;restore the caller's ES register
     POP       DS                            ;restore the caller's DS register
     POP       DX                            ;restore the caller's DX register
     POP       CX                            ;restore the caller's CX register
     POP       BX                            ;restore the caller's BX register
     POP       AX                            ;restore the caller's AX register
     RET                                     ;return to caller BDOS

EXIT_PROC  ENDP
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                    Driver's MSDOS Command Code Routines
;
;
; The MSDOS Command Code Routines do nothing other than exit with the
; RETURN_STATUS word in the request header set to done without errors.
;
MSDOS_MEDIA_CHECK:                      ;MSDOS Command Code # 1
     NOP                                ;code to process MEDIA CHECK
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_BUILD_BPB:                        ;MSDOS Command Code # 2
     NOP                                ;code to process BUILD BPB
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_IOCTL_INPUT:                      ;MSDOS Command Code # 3
     NOP                                ;code to process IOCTL INPUT
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_INPUT:                            ;MSDOS Command Code # 4
     NOP                                ;code to process INPUT
     JMP       EXIT                     ;jump to the exit routine

MSDOS_NON_DSTRV_INPUT:                  ;MSDOS Command Code # 5
     NOP                                ;code to process NON DSTRV INPUT
     JMP       EXIT                     ;jump to the exit routine

MSDOS_INPUT_STATUS:                     ;MSDOS Command Code # 6
     NOP                                ;code to process INPUT STATUS
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_INPUT_FLUSH:                      ;MSDOS Command Code # 7
     NOP                                ;code to process INPUT FLUSH
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_OUTPUT:                           ;MSDOS Command Code # 8
     NOP                                ;code to process OUTPUT
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_OUTPUT_WITH_VERIFY:               ;MSDOS Command Code # 9
     NOP                                ;code to process OUTPUT W/VERIFY
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_OUTPUT_STATUS:                    ;MSDOS Command Code #10
     NOP                                ;code to process OUTPUT STATUS
     JMP       EXIT                     ;jump to the EXIT routine

MSDOS_OUTPUT_FLUSH:                     ;MSDOS Command Code #11
     NOP                                ;code to process OUTPUT FLUSH
     JMP       EXIT                     ;jump to the EXIT routine
```

```
MSDOS_IOCTL_OUTPUT:                              ;MSDOS Command Code #12
     NOP                                         ;code to process IOCTL OUTPUT
     JMP       EXIT                              ;jump to the EXIT routine

MSDOS_INITIALIZE:                                ;MSDOS Command Code #0
     LDS       BX,CS:[HEADER_POINTER]            ;point DS:BX to the request header
     MOV       WORD PTR DS:[BX.TRANSFER_ADDRESS],OFFSET MSDOS_INITIALIZE
     MOV       WORD PTR DS:[BX.TRANSFER_ADDRESS+2],CS
     JMP       EXIT


;
DRIVER  ENDP
;
CODE    ENDS
        END       START
```

AGIOS: I/O CONTROL OF THE CON DEVICE

The Alpha/Graphic Input/Output System (AGIOS)

The Alpha/Graphic I/O system is a set of functions which can be called from application programs. The functions help programmers manipulate the HP 150 video display, control the touchscreen, and handle the keyboard. The AGIOS functions provide a high level interface to these entities sparing the programmer of much tedium which may otherwise be necessary. The price which is sometimes paid for this convenience is performance. For this reason, Section 7, Programming the HP 150, includes some information allowing programmers access to the alpha and graphics memories in a direct manner.

Accessing the AGIOS

The Alpha/Graphic I/O System is implemented as a set of functions which are really MS-DOS System Function Call sub-functions. MS-DOS includes a function, I/O control for devices (function 44H). This function enables an IOCTL device command to an open device to be performed. The IOCTL function can be implemented by a device driver to allow control-type manipulation of the device, rather than read/write oriented operations. The HP 150 console device "CON" implements the AGIOS function set through its IOCTL call.

Most high level programming languages on the HP 150 do not allow direct manipulation of the 8088 registers. To perform an MS-DOS System Function and hence, an AGIOS function, the registers must be loaded with function-dependent data. As a result, AGIOS function calls from high level languages are usually made through assembly language subroutines combined with the higher level code at link time.

The following code is an example of an AGIOS function call written in assembly language. It is the "Execute Two Character Escape Sequence" AGIOS function.

This AGIOS function call requires a three byte buffer which contains the function code followed by the "J" parameter.

```
          +------+-------+-------+
BUFF      | 16   |   0   |   J   |
          +------+-------+-------+
Byte:        +0      +1      +2
```

The program segment to accomplish the clear display would look something like this in assembler:

```
clrscrn   LABEL NEAR
          MOV   AX,4403H          ;I/O Control Write
          MOV   BX,1              ;Console Handle, always = 1
          MOV   CX,3              ;buff length (3 in this case)
          MOV   DX,OFFSET buff    ;and Offset
          INT   21H               ;MS-DOS call
          RET
buff      DB    16,0,'J'
```

The above example lacks elegance and structure typical of good programming practice. For example, the buffer could reside in the data segment and be accessible such that other escape sequences could be programmed. Or the function dependent argument ("J" in this case) could be passed on the stack. Nor does the above routine do any error checking. If an MS-DOS error occurs when you make an AGIOS function call, the "carry flag" is set according to the standard MS-DOS procedures. If an AGIOS error occurs, the AX register contains a non-zero value. AGIOS errors can occur when MS-DOS errors do not occur. This means that the carry flag is not set but the AX register contains a non-zero character. Therefore, you need to check both of these indicators after a function call to determine if the operation was completed successfully.

For more information on making AGIOS and MS-DOS calls from high level languages, together with some more generalized examples, refer to Section 7, Programming the HP 150.

BIOS AND ITS DEVICES

Introduction

The HP 150's Basic I/O System (BIOS) consists of a set of twelve modules that are responsible for interacting with Microsoft's Basic Disc Operating System (BDOS) on one side, and with the HP 150 firmware routines on the other. The interface between the BDOS and the BIOS is specified by Microsoft, and the HP 150's BIOS modules implement this interface. (See "Calling a Device Driver" earlier in this section.) However, to perform the actual I/O operations they rely heavily on the services provided by the firmware. Much of what these modules do is to take function calls handed down from the BDOS, translate them into appropriate firmware calls, and call the firmware routines to execute the I/O function.

CONSOLE DEVICE

Device Names: CON

Device Functions Implemented:

0	INIT	Not implemented
1	MEDIA CHECK	Not implemented
2	BUILD BPP	Not implemented
3	IOCTL INPUT	Reserved - returns error
4	INPUT (read)	Valid
5	NON-DESTRUCTIVE INPUT NO WAIT	Valid
6	INPUT STATUS	Not implemented
7	INPUT FLUSH	Valid
8	OUTPUT (write)	Valid
9	OUTPUT (write) WITH VERIFY	Valid
10	OUTPUT STATUS	Not required by firmware
11	OUTPUT FLUSH	Valid
12	IOCTL OUTPUT	Valid - AGIOS

SERIAL DATA COMMUNICATIONS DEVICES

Device Names:

COM, COM1, COM2, PRN, LST, AUX, PLT, LPT1, LPT2, LPT3

Device Functions Implemented:

0	INIT	Not implemented
1	MEDIA CHECK	Not implemented
2	BUILD BPP	Not implemented
3	IOCTL INPUT	Valid - returns on buffer empty
4	INPUT (read)	Valid
5	NON-DESTRUCTIVE INPUT NO WAIT	Valid
6	INPUT STATUS	Not implemented
7	INPUT FLUSH	Valid
8	OUTPUT (write)	Valid
9	OUTPUT (write) WITH VERIFY	Valid
10	OUTPUT STATUS	Valid
11	OUTPUT FLUSH	Not implemented
12	IOCTL OUTPUT	Valid

TIME DEVICE

Device Name:

CLOCK

Device Functions Implemented:

0	INIT	Not implemented
1	MEDIA CHECK	Not implemented
2	BUILD BPP	Not implemented
3	IOCTL INPUT	Reserved - returns an error
4	INPUT (read)	Valid
5	NON-DESTRUCTIVE INPUT NO WAIT	Not implemented - returns busy flag
6	INPUT STATUS	Not implemented
7	INPUT FLUSH	Not implemented
8	OUTPUT (write)	Valid
9	OUTPUT (write) WITH VERIFY	Valid
10	OUTPUT STATUS	Not implemented
11	OUTPUT FLUSH	Not implemented
12	IOCTL OUTPUT	Not implemented

INTEGRAL PRINTER DEVICE

Device Names:

INT, PRN, AUX, LST, LPT1, LPT2, LPT3

Driver Functions Implemented:

0	INIT	Not implemented
1	MEDIA CHECK	Not implemented
2	BUILD BPP	Not implemented
3	IOCTL INPUT	Reserved - returns an error
4	INPUT (read)	Valid
5	NON-DESTRUCTIVE INPUT NO WAIT	Not implemented - returns an error
6	INPUT STATUS	Not implemented
7	INPUT FLUSH	Not implemented
8	OUTPUT (write)	Valid
9	OUTPUT (write) WITH VERIFY	Valid
10	OUTPUT STATUS	Valid
11	OUTPUT FLUSH	Not implemented
12	IOCTL OUTPUT	Not implemented

DISC, HPIB PRINTER, HPIB PLOTTER DEVICES

Device Names:

A:....L:, AUX, PRN, LST, PLT, LPT1, LPT2, LPT3

Driver Functions Implemented:

0	INIT	Not implemented
1	MEDIA CHECK	Not implemented
2	BUILD BPP	Not implemented
3	IOCTL INPUT	Reserved - returns an error
4	INPUT (read)	Valid
5	NON-DESTRUCTIVE INPUT NO WAIT	Not implemented - returns busy flag
6	INPUT STATUS	Not implemented
7	INPUT FLUSH	Not implemented
8	OUTPUT (write)	Valid
9	OUTPUT (write) WITH VERIFY	Valid
10	OUTPUT STATUS	Valid
11	OUTPUT FLUSH	Not implemented
12	IOCTL OUTPUT	Not implemented

HPIBDEV DEVICE

Device Name:

HPIBDEV

Driver Functions Implemented:

```
 0  INIT ......................... Not implemented
 1  MEDIA CHECK .................. Not implemented
 2  BUILD BPP .................... Not implemented
 3  IOCTL INPUT .................. Special HP data returns
 4  INPUT (read) ................. Not implemented
 5  NON-DESTRUCTIVE INPUT NO WAIT  Not implemented - returns busy flag
 6  INPUT STATUS ................. Not implemented
 7  INPUT FLUSH .................. Not implemented
 8  OUTPUT (write) ............... Valid
 9  OUTPUT (write) WITH VERIFY ... Valid
10  OUTPUT STATUS ................ Valid
11  OUTPUT FLUSH ................. Not implemented
12  IOCTL OUTPUT ................. Valid - pass through of template & data
```

THE CONFIG.SYS FILE

In many cases, there are installation-specific configurations of the operating system that are needed to be set up at boot time. The user need not re-build the DOS (BDOS) to include special drivers or to include a particular number of device drivers. Drivers for additional devices may be included as separate entities. See the discussion of devices earlier in this chapter. The configuration file allows a user to configure his system without extra work.

The configuration file is simply an ASCII file that has certain commands for the operating system initialization task.

During operating system initialization (booting) a long jump to the SYSINIT routine in the BIOS SYSINIT module is made. This module (supplied with the operating system from Microsoft) will initialize the DOS and read the configuration file CONFIG.SYS, if it exists, to perform device installation and various other user settable things.

The following are a list of commands for the configuration file CONFIG.SYS:

BUFFERS = <*number*>
This is the number of additional sector buffers to add to the system list. The effect of several BUFFERS commands is to allocate a series of buffers. The default is BIOS specific, equal to 16 in versions A.01.02 and A.01.06. The minimum value allowed for BUFFERS is one. With Winchester disc based systems, performance may be improved by increasing the buffer or size so as to allow the entire disc FAT (File Allocation Table) into buffer memory. As such, the operating system can access the entire FAT without going to disc.

FILES = <*number*>
This is the number of open files that the XENIX system calls can access. The default is BIOS specific, equal to 20 in versions A.01.02 and A.01.06. If a number less than or equal to five is specified, the command is ignored.

DEVICE = <*filename*>
This installs the device driver in <filename> into the system list.

BREAK = <*ON or OFF*>
If ON is specified (the default is OFF), a check for 'C at the console input will be made every time the system is called. ON improves the ability to abort programs over previous versions of the DOS.

SWITCHAR = <*char*>
Causes the DOS to return <char> as the current switch designator character when the DOS call to return the switch character is made. Default is '/'. Note that the setting of SWITCHAR may effect characters used on the SHELL line (this is true of COMMAND.COM).

AVAILDEV = <*TRUE or FALSE*>
The default is TRUE which means both /dev/<dev> and <dev> will reference the device <dev>. If FALSE is selected, only /dev/<dev> refers to device <dev>,

<dev> by itself means a file in the current directory with the same name as one of the devices.

SHELL = <*filename*>
This begins execution of the shell (top-level command processor) from <filename>. Note that the parameters on this line are shell dependent.

A typical configuration file might look like this:

```
BUFFERS = 10
FILES = 10
DEVICE = /bin/network.sys
BREAK = ON
SWITCHAR = -
SHELL = pamcode.exe root
```

DISC FORMAT AND DIRECTORY STRUCTURE

Physical Disc Format

HP 150 disc media are partitioned physically into "tracks" which in turn are each partitioned into "sectors". Each sector is a physical portion of a track commonly containing 256 bytes of information.

Sectors are numbered 0, 1, 2, 3, ... and so on. This is not to say that sector 1 follows sector 0 physically on the disc. Sector "staggering" is employed to improve disc read and write efficiency.

The disc drive takes care of logical to physical mapping and as such the programmer need only be concerned with logical sector numbering, above.

Disc Media Storage Capacity

The following table shows the storage capacity in sectors and number of kilobytes for different types of disc drives available on the HP 150.

Media Type	Total Sectors	Total KB
3-1/2" Single Sided Microfloppy	1,056	264
3-1/2" Double Sided Microfloppy	1,385 *	709
5-1/4" Double Sided Minifloppy	1,056	264
8" Floppy (HP Format)	4,500	1,125
8" Floppy (IBM Format)	2,002 **	250
"5MB" Winchester	18,848	4,712
"10MB" Winchester	37,820	9,455
"15MB" Winchester	56,730	14,182

* Sector size is 512 bytes.
** Sector size is 128 bytes.
All other media have 256 byte sectors.

Disc Sector Allocation

Beginning with logical sector number 0, several contiguous sectors are reserved for system use, that is they do not contain file data. The sector map of a disc looks like this:

Sector Number	Contains
0	Header Record
1	Boot sector for operating system disc. FF's for non-O.S. disc.
2-XX	File Allocation Table (FAT) Number 1
XX-XX	File Allocation Table (FAT) Number 2
XX-XX	Disc Directory
XX - Last	Disc Data Area

NOTE: The number of FAT, DIRECTORY and DATA sectors varies from media type to media type.

Header Record

The header record contains disc dependent data. It is always the first logical disc record. The header record is structured as follows.

ENTRY REFERS TO	DISC TYPE			
	3" & 5" single	3" dbl. sided	RAM disc	8"
EBH, 1CH,90H (3 Bytes)	All discs contain this information			
"HP150 " (8 Bytes)	All discs contain this information			
bytes per sector (W)	256	512	512	256
sectors per cluster* (B)	4	2	1	16
reserved sectors (W)	2	2	2	2
number of FATs (B)	2	2	1	1
number of DIR entries (W)	128	128	128	256
total sectors (W)	1,056	1,385	**	4,500
media type = FAH (B)	All discs use this value			
sectors per FAT (W)	3	3	3	3
sectors per track (W)	16	8	NA	30
number of heads (W)	2	2	NA	2
# of hidden sectors = 00H (W)	All discs use this value			

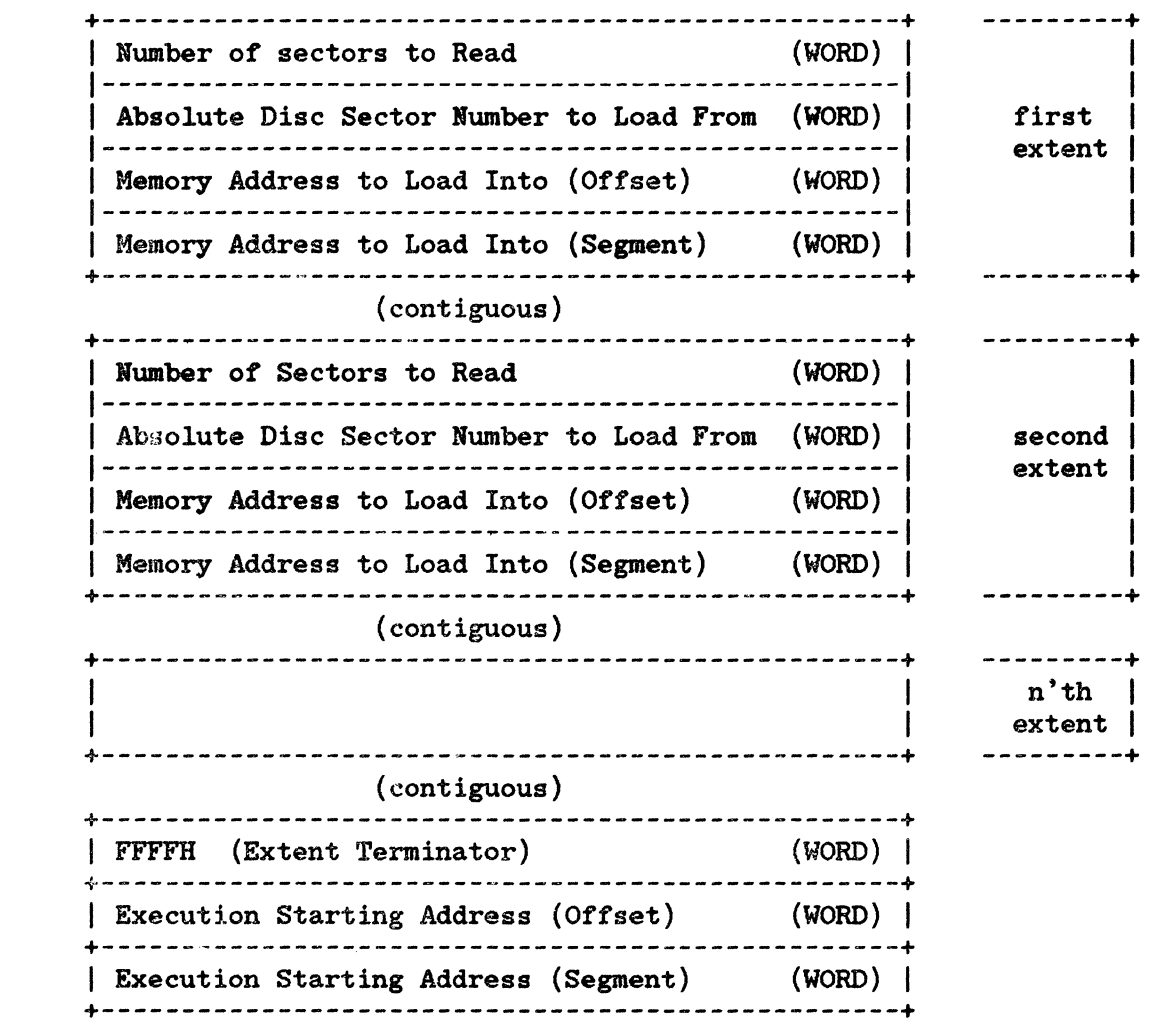
ENTRY REFERS TO	DISC TYPE			
	5 MB	10 MB	15 MB	IBM 8"
EBH, 1CH,90H (3 Bytes)	All discs contain this information			
"HP150 " (8 Bytes)	All discs contain this information			
bytes per sector (W)	256	256	256	128
sectors per cluster* (B)	16	16	16	8
reserved sectors (W)	2	2	2	1
number of FATs (B)	2	2	2	2
number of DIR entries (W)	1,024	1,024	1,024	68
total sectors (W)	8,848	37,820	56,730	2,002
media type = FAH (B)	All discs use this value			
sectors per FAT (W)	9	15	21	6
sectors per track (W)	31	31	31	26
number of heads (W)	4	4	6	1
# of hidden sectors = 00H (W)	All discs use this value			

* Also referred to an an allocation unit.
** Size specified in the DEVCONFG utility.

Boot Sector

The boot sector holds an index to other sectors which contain the operating system code (MS-DOS). During system initialization, the firmware reads in the code from the sectors indexed by the boot sector. If the disc is a non-operating system disc, this sector contains FF's.

The format of the boot sector of a disc containing an operating system is as follows.



Note: The second through n'th extents may not exist.

File Allocation Table (FAT)

DISC CLUSTERS
Disc sectors are grouped together into what are known as "clusters". On a 3 1/2" flexible disc, there are four sectors per cluster. Files are assigned space on the disc in increments of 1 cluster. That is for a 3 1/2" disc files are built in one kilobyte pieces.

The File Allocation Table is an area partitioned into many one and one half byte (12 bit) entries, each representative of a particular cluster. A file's directory entry contains a starting cluster number. The FAT entry corresponding to this cluster will point to another cluster, being the second cluster of the file. The FAT entry corresponding to this second cluster will point to a third cluster and so on. That is, the FAT entries describe a linked list of clusters containing the file.

FAT STRUCTURE The File Allocation Table always begins on the first section after the reserved sectors. If the FAT is larger than one sector, the sectors are contiguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers whenever needed (open, read, write, etc.). The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk. The first two FAT entries map a portion of the directory; these FAT entries indicate the size and format of the disk.

The second and third bytes currently always contain FFH.

The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated. The first free cluster found will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

Each FAT entry contains three hexadecimal characters:

000	If the cluster is unused and available.
FF7	The cluster has a bad sector in it. MS-DOS will not allocate such a cluster. CHKDSK counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.
FF8-FFF	Indicates the last cluster of a file.
XXX	Any other characters that are the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

HOW TO USE THE FILE ALLOCATION TABLE The following information is included for system programmers who wish to write installable block device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver is then responsible for locating the logical sector on disk. Programs must use the MS-DOS file management function calls for accessing files; programs that access the FAT are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

1. Multiple the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).

2. The whole part of the product is an offset into the FAT pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

3. Use a MOV instruction to move the word at the calculated FAT offset into a register.

4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

5. If the resultant 12 bits are FF8H-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.

2. Multiply the result by the number of sectors per cluster.

3. Add to this result the logical sector number of the beginning of the data area.

MS-DOS Disc Directory

The disc directory is a list of all files residing on the disc. The FORMAT program builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as file by MS-DOS, there is no limit to the number of files they may contain.

All directory entries are 32 bytes in length, and are in the following format (note that byte offsets are in hexadecimal):

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. The first byte of this field indicates the file status as follows:

00H The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.

2EH The entry is for a directory. If the second byte is also 2EH, then the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are all spaces, and the cluster field contains the cluster number of this directory.

E5H The file was used, but it has been erased.

Any other character is the first character of a filename.

8-0A Filename extension

0B File attribute. The attribute byte is mapped as follows (values are in hexadecimal):

01 File is marked read-only. An attempt to open the file for writing using the Open File system call (Function Request 3DH) results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the Delete File system call (13H) or Delete a Directory Entry (41H) will also fail.

02 Hidden file. The file is excluded from normal directory searches.

04 System file. The file is excluded from normal directory searches.

08 The entry contains the volume label in the first 11 bytes. The entry contains no other usable information (except date and time of creation), and may exist only in the root directory.

10 The entry defines a sub-directory, and is excluded from normal directory searches.

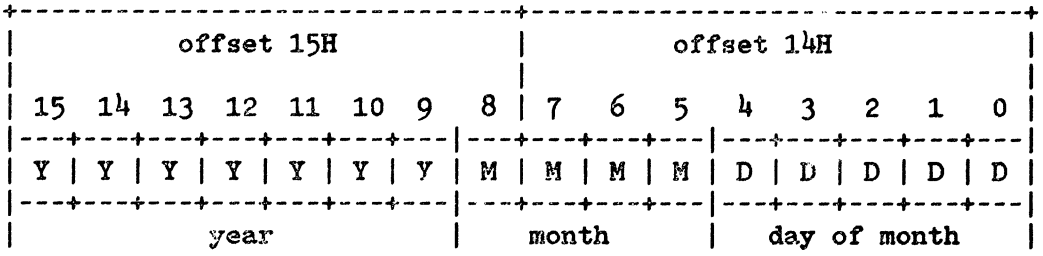
20 Archive bit. The bit is set to "on" whenever the file has been written to and closed.

Note: The system files (IO.SYS and MSDOS.SYS) are marked as read only, hidden, and system

files. Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the Change Attributes system call (Function Request 43H).

0C-15 Reserved.

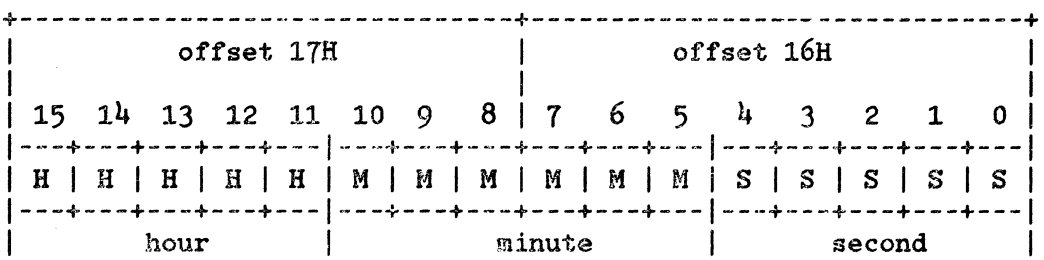
16-17 Time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:



where:

H is the binary number of hours (0-23)
M is the binary number of minutes (0-59)
S is the binary number of two second increments

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:



where:

Y is 0-119 (1980-2099)
M is 1-12
D is 1-31

1A-1B Starting cluster; the cluster number of the first cluster in the file.

Note that the first cluster for data space on all disks is cluster 002.

The cluster Number is stored with the least significant byte first.

NOTE

Refer to section "How to Use the File Allocation Table," for details about converting cluster numbers to logical sector numbers.

1C-1F File size in bytes. The first word of this four-byte field is the low order part of the size.

SYSTEM FIRMWARE

SECTION 6

The HP 150 system contains a large amount of system management code located in Read Only Memory. In general, the firmware is not designed to be utilized by system programmers. The operating system BIOS serves as the interface to firmware located functionality.

The HP 150 firmware is a rather complex multi-tasking operating system in itself. MS-DOS runs on the firmware as a single task from the firmwares perspective. Essentially all of the terminal functionality of the HP 150 is implemented in the firmware. The firmware implements much of the BIOS device driver functionality in addition to the terminal personality.

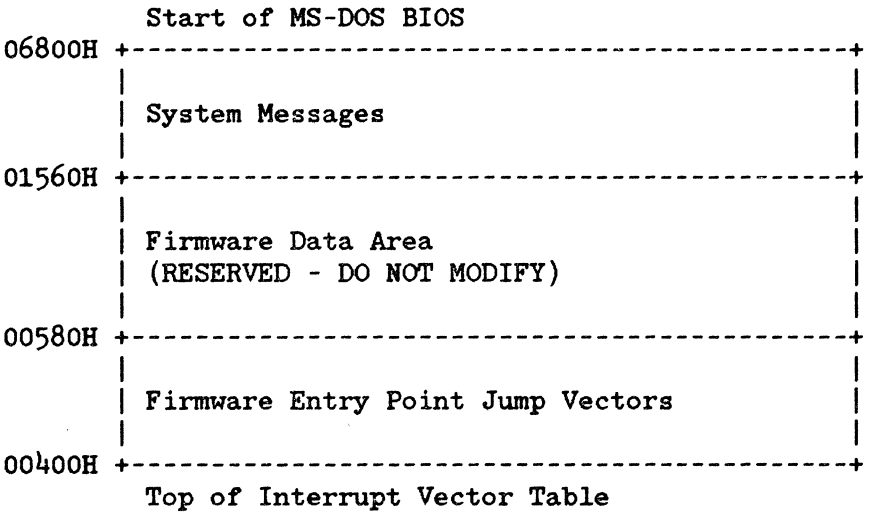
This section details the firmware's useage of the 26 K Byte of system RAM which it claims, and includes a table of the firmware entry points, vectors located within a portion of that memory.

CONTENTS

FIRMWARE MEMORY MAP

RAM

Portion of the system RAM memory is reserved for the use of the firmware. The following shows how that address space is allocated.



FIRMWARE ENTRY POINT JUMP VECTORS

The firmware is highly structured, consisting of many tasks drivers, and software processors. Long calls may be made to these entities through vectors located in an area of firmware RAM. In general, each entry point detailed in the table below has a defined set of functions and parameter passing conventions for those functions. Programs should not call these functions, except when implementing operations as described in section 7 of this manual, Programming the HP 150. Access to the firmware entry points (located in ROM) should at all times be made through the following jump vectors.

Address	Vector to:	Length (bytes)
00400H	System timer for MS-DOS and applications	4
00420H	THP_ENTRY	5
00425H	DHP_ENTRY	5
0042AH	DHP_DRIVER	5
0042FH	MMP_ENTRY	5
00434H	STP_ENTRY	5
00439H	IWP_ENTRY	5
0043EH	PHP_ENTRY	5
00443H	PAP_ENTRY	5
00448H	PAP_PARSE	5
0044DH	PAP_PERFORM	5
00452H	DTP_ENTRY	5
00457H	VCP_ENTRY	5
0045CH	CNP_ENTRY	5
00461H	DMP_ENTRY	5
00466H	WMP_ENTRY	5
0046BH	KBP_ENTRY	5
00470H	DCP_ENTRY	5
00475H	RSP_ENTRY	5
0047AH	UTP_ENTRY	5
0047FH	NVRP_ENTRY	5
00484H	GSP_ENTRY	5
00489H	BMP_ENTRY	5
0048EH	STGP_ENTRY	5
00493H	HPSP_ENTRY	5
00498H	SKP_ENTRY	5
0049DH	SWP_ENTRY	5
004A2H	SLP_ENTRY	5
004A7H	GRP_ENTRY	5
004ACH	GRP_TKPARSE	5
004B1H	TSP_ENTRY	5
004B6H	TPM_SKP_ENTRY	5
004BBH	IHP_ENTRY	5
004C0H	OHP_ENTRY	5
004C5H	OHP_OPTION	5
004CAH	VAU_EXTERNAL_DRIVER	5
004CFH	WMP_ENTRY	5
004D4H	KBT_ENTRY1	5
004D9H	KBT_ENTRY2	5

PROGRAMMING THE HP 150

SECTION 7

This section is intended as an aid to programmers wishing to write applications programs to run on the HP 150. The HP 150 includes many system level functions intended to ease the complexity of programs running on the machine. Some of the examples included in this section demonstrate the use of these facilities.

In some cases the functions provided, particularly in the area of video interfacing, may not provide acceptable performance levels for your application. For this reason the internal video structure of the HP 150 is described. The HP 150's built-in capabilities in the area of data communications are extensive. Information is included in this section to allow a programmer to take advantage of some of these capabilities. Finally, the HPIB interface on the HP 150 is described from an applications program standpoint. The discussion and corresponding examples show how to drive this interface and as such extend the capabilities of the HP 150 even further.

CONTENTS

GENERATING ESCAPE SEQUENCES

From the Microsoft C Language for example, a formatted print function (from the Microsoft C Library) may be used to send an escape sequence to the console escape sequence processor. Note that in the following example the string is terminated by cariage return and linefeed characters to cause the console output buffer to be "flushed", that is have the information actually processed.

```
#define ESC 0x1B

main()
{
   printf ( "%c&a12c2Y\r\n", ESC );
}
```

The above example moves the cursor to row 2, column 12 in the currently displayed screen window.

CAUTION

Because AGIOS calls are processed immediatly the call is issued and escape sequences are not processed until the console buffer is flushed, mixing the two without flushing the console buffer prior to making the AGIOS call may result in the AGIOS call being processed first.

PERFORMING MS-DOS SYSTEM FUNCTION CALLS

From the Microsoft C Language for example, MS-DOS System Function calls are most easily made through an assembly language routine combined with the C object at link time. The following is an example of a general purpose System Function call written in assembly language. It is actually an assembly language coding of a Microsoft C compatible function. The function, when called with the appropriate parameters, can invoke almost any MS-DOS System Function. Parameters are passed on the stack. Note that since parameter passing conventions between routines vary from language to language, this function is particular to Microsoft C.

```
;-------------------------------------------------------------------------
;
;  doscall ( ax, bx, cx, dx )
;     int ax, bx, cx, dx;
;
;  Issues an MS-DOS System Function Call with registers passed in using
;  Microsoft C parameter passing convention and registers passed out in
;  global four word variable.
;
;  Parameters in:
;     ax - Value to put in register AX
;     bx - Value to put in register BX
;     cx - Value to put in register CX
;     dx - Value to put in register DX
;
;  Return value:
;     1 (boolean TRUE)  - Carry flag was set by function (error for some f'ns)
;     0 (boolean FALSE) - Carry flag not set by function (no error)
;
;  Parameters out:
;     Registers AX, BX, CX, and DX are returned as four contiguous words
;     in global variable 'dc_ret' (which is defined by this routine)
;     Word 1 - AX register returned by function
;     Word 2 - BX register returned by function
;     Word 3 - CX register returned by function
;     Word 4 - DX register returned by function

TRUE        EQU   1                 ;Boolean true for return value
FALSE       EQU   0                 ;Boolean false for return value

dc_args     STRUC                   ;Stack structure for passed parameters
            DW    ?                 ;Callers BP
            DW    ?                 ;Return address
ax_in       DW    ?                 ;'ax'
bx_in       DW    ?                 ;'bx'
cx_in       DW    ?                 ;'cx'
dx_in       DW    ?                 ;'dx'
dc_args     ENDS
```

```
ret_args    STRUC                   ;Structure for returned parameters
ax_out      DW    ?                 ;Returned AX word
bx_out      DW    ?                 ;Returned BX word
cx_out      DW    ?                 ;Returned CX word
dx_out      DW    ?                 ;Returned DX word
ret_args    ENDS

DGROUP      GROUP DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:GROUP

            PUBLIC dc_ret
dc_ret      ret_args <0,0,0,0>      ;Global variable 'dc_ret' declaration

DATA        ENDS

PGROUP      GROUP PROG
PROG        SEGMENT WORD PUBLIC 'PROG'
            ASSUME CS:PGROUP, DS:DGROUP

            PUBLIC doscall
doscall     PROC  NEAR
            PUSH  BP                ;Save BP and set it up as a pointer into
            MOV   BP,SP             ;the stack to retrieve passed arguments
            MOV   AX,[BP].ax_in     ;Copy AX parameter from stack
            MOV   BX,[BP].bx_in     ;Copy BX parameter from stack
            MOV   CX,[BP].cx_in     ;Copy CX parameter from stack
            MOV   DX,[BP].dx_in     ;Copy DX parameter from stack
            INT   21H               ;Invoke MS-DOS System Function
            MOV   DGROUP:dc_ret.ax_out,AX     ;AX word for return
            MOV   DGROUP:dc_ret.bx_out,BX     ;BX word for return
            MOV   DGROUP:dc_ret.cx_out,CX     ;CX word for return
            MOV   DGROUP:dc_ret.dx_out,DX     ;DX word for return
            MOV   AX,FALSE          ;Assume no error
            JNC   dcall_end
            MOV   AX,TRUE
dcall_end   LABEL NEAR
            POP   BP                ;Restore BP
            RET
doscall     ENDP

PROG        ENDS

            END
```

As an example of a C program performing an MS-DOS Function Call consider the following:

```
#define DUMMY 0

struct dc_ret_tem  {   /* 'doscall' return parameter structure */
                   char al;
                   char ah;
                   char bl;
                   char bh;
                   char cl;
                   char ch;
                   char dl;
                   char dh;
                   };

extern struct dc_ret_tem dc_ret;

main ()          /* Prints the drive letter for the current default disc */
{
char drive;

   doscall ( 0x1900, DUMMY, DUMMY, DUMMY );   /* Current Disc System F'n */
   if ( dc_ret.al < 12 )
      drive = dc_ret.al + 'A';
   else
      drive = '?';
   printf ( "Current Disc is %c\r\n", drive );
}
```

This program invokes MS-DOS System Function Call 19H (Current Disc) which returns a number in AL indicating the current default disc drive. The appropriate drive letter is printed.

MAKING AGIOS FUNCTION CALLS

From the Microsoft C Language for example, AGIOS Function calls are most easily made through an assembly language routine combined with the C object at link time. AGIOS calls, being a special case of the ***doscall*** function previously described, could be performed using that assembly language routine. However for the sake of efficiency another routine is described. It is actually an assembly language coding of a Microsoft C compatible function. The function, when called with the appropriate parameters, can be used to invoke any AGIOS function. Parameters are passed on the stack. Note that since parameter passing conventions between routines vary from language to language, this function is particular to Microsoft C. Refer to Section 8, AGIOS Function Call Reference, for a directory of AGIOS functions.

```
;-------------------------------------------------------------------------
;  int agios ( buf_add, length )
;  int buf_add, length
;
;  Performs an AGIOS function call with parameters passed using Microsoft
;  C language parameter passing conventions. Calls MSDOS function number 044H
;  (I/O Control For Devices) with sub-function 003H (Write to Device Control
;  Channel) using CON (console) handle.
;
;  Parameters in:
;     buf_add - The address (offset - assumes same data segment) of the AGIOS
;               parameter buffer for the call.  Specifies the particular
;               AGIOS function to be performed and other pertinent data for
;               the function.
;     length  - Specifies the length of the above parameter buffer in bytes.
;
;  Return value:
;          0 - AGIOS function executed with no error.
;     Non-0 - AGIOS function executed with error.  Value returned is the
;               status code returned from the function specified in the
;               parameter buffer (see AGIOS function documentation).
;
;  Parameters out:  none

PGROUP      GROUP  PROG
PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME CS:PGROUP

CONIN       EQU    0              ;Standard MS-DOS console handle

agios_args  STRUC                            ;Stack structure for passed args.
            DW     ?                         ;Callers BP
            DW     ?                         ;Return address
buf_add     DW     ?                         ;Parameter buffer address offset
buf_len     DW     ?                         ;Number of bytes in parm buffer
agios_args  ENDS
```

```
            PUBLIC agios
agios       PROC   NEAR                ;Write control string to AGIOS
            PUSH   BP                  ;Save callers BP
            MOV    BP,SP               ;Initialize base pointer
            MOV    AX,04403H           ;MS-DOS System Function 44H, Sub-
                                       ;  Function 3
            MOV    BX,CONIN            ;Device handle
            MOV    CX,[BP],buf_len     ;Get length of parameter buffer
            MOV    DX,[BP],buf_add     ;Get buffer address
            INT    21H                 ;Call MS-DOS
            POP    BP                  ;Restore callers BP
            RET                        ;Return to caller
agios       ENDP

PROG        ENDS

            END
```

Since the above assembly language function can be used to perfrom any AGIOS function, you may wish to put a shell around it for any particular AGIOS functions called. This will be done for Function 17, Position Cursor in the following example.

First, declare the parameter buffer structure and data area:

```
struct   agios_pc_tem   {
                        unsigned int funct_num;    /* function number */
                        char         mode;         /* mode */
                        unsigned int column;       /* column number */
                        unsigned int row;          /* row number */
                        };
```

The above serves as a template for the parameter buffer particular to the Position Cursor function.

Now two constants, the parameter buffer length and the Position Cursor function number are defined:

```
#define PC_FLEN          sizeof(struct agios_pc_tem)  /* Function length */
#define PC_FNUM          17                           /* Function number */
```

The following statement actually reserves a data storage area for the parameter buffer.

```
struct  agios_pc_tem  agios_pc_par = { PC_FNUM, 0, 0, 0, };
```

The variable 'ret_code' is used for the integer value returned by an AGIOS function call. The value indicates success or failure of the function call.

```
int ret_code;                          /* Function call return value */
```

Now the cursor positioning function is defined:

```
cursor ( row, column )

unsigned int row;
unsigned int column;

{
   agios_pc_par.row    = row;
   agios_pc_par.column = column;
   ret_code = agios ( &agios_pc_par, PC_FLEN ); /* perform the AGIOS call */
}
```

Note that the *mode* parameter has not been defined. This would be done in most cases only once outside the above routine. The mode parameter is defined and the above function called with the following statements:

```
   agios_pc_par.mode   = 0x99;      /* row and column parameters are both valid
                                       positive absolute window values */
   cursor (2, 12);
```

ALPHANUMERIC DISPLAY INTERFACING

Alphanumeric Video RAM Structure

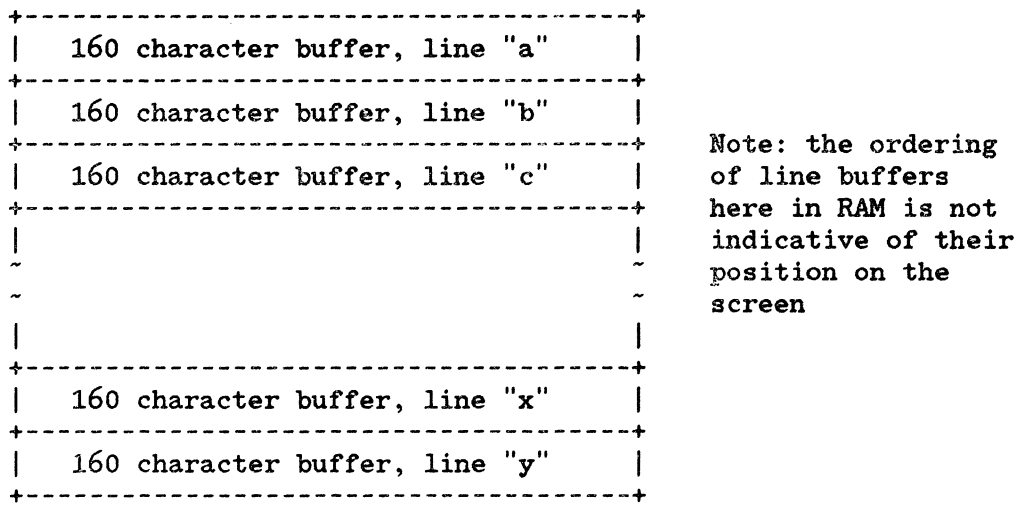
The following information describes the internal architecture of the HP 150 alphanumeric video structure. It provides programmers with a means of bypassing the console driver (CON: device) and its AGIOS function subset for the purpose of putting alphanumeric information to the screen.

The AGIOS function set contains some powerful screen manipulation capabilities and is the preferred method of accessing the alpha (and graphics) video memory. In situations where these functions provide adequate functionality and performance they should be used.

The HP 150 has 48 lines of display normally available to applications through console outputs and the AGIOS function calls. The following information applies only to the 27 lines (24 text plus two softkey plus one status) forming the physical screen as it appears at any given time.

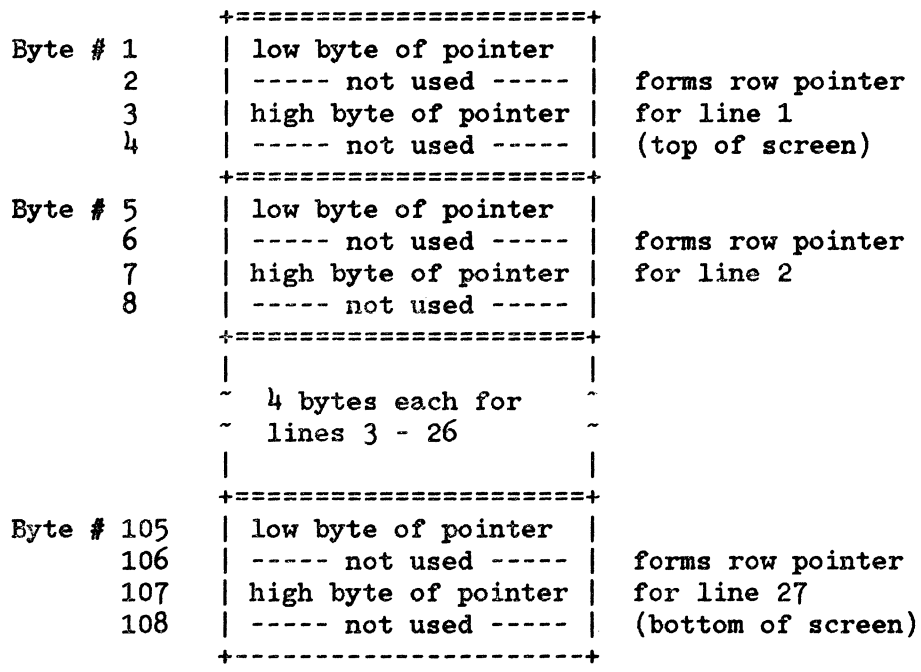
Each of the 27 lines of alphanumeric text displayed on the screen are represented by 80 consecutive words (pairs of bytes) in RAM. A table (video row pointer table) indexes the address of the first byte in each line. This table is read by the HP 150 video control hardware directly, and the video control chip DMAs into the appropriate area of memory accordingly. Note that consecutively displayed lines on the screen do not necessarily have their storage areas in a corresponding order in RAM.

ALPHA VIDEO BUFFER FORMAT



VIDEO ROW POINTER TABLE FORMAT

The row pointer table also located in alphanumeric video RAM contains 27 consecutive pointers indexing the first character of each line in the alpha line buffer. That is each pointer indexes the first byte in the above buffer. Note that the pointers are arranged line 1 through line 27 as described in the following diagram.



ROW POINTER FORMULATION FROM ABOVE TABLE

	high byte of pointer							low byte of pointer *							
bit number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
represents	***		14 bit word offset from alpha RAM origin (D0000H)													

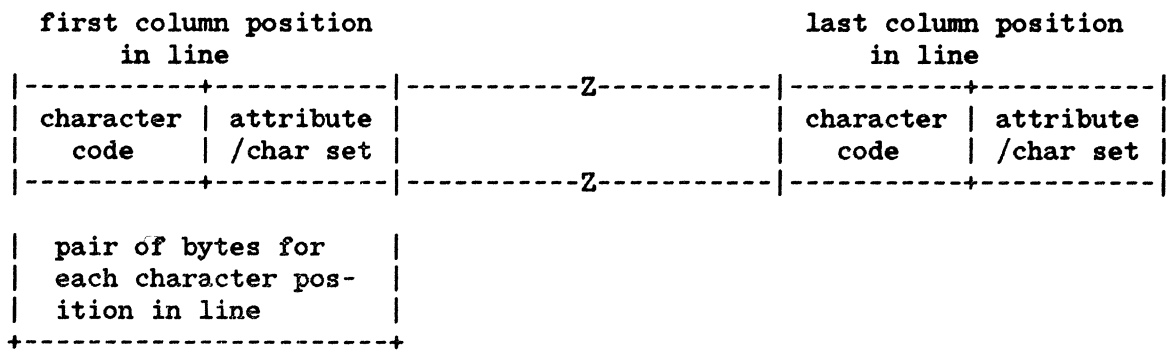
* The least significant bit (A0) must be a zero (0).

*** These bits define the double height and width characteristics of the referenced row. Note that double high/double wide characters are not supported by the HP 150 firmware, software or hardware.

D15	D14		
0	0	Single height, single width (standard)	
0	1	Single height, double width	\ not
1	0	Double height top half, double width	| supported
1	1	Double height bottom half, double width	/ (will not work)

LINE BUFFER FORMAT

The diagram below describes the format of each display line's RAM buffer.



ENHANCEMENT/CHARACTER SET BYTE STRUCTURE

D7	D6	D5	D4	D3	D2	D1	D0
security	half bright	underline	inverse	blink	character set code		

NORMAL ROMAN	0	0	0
ROMAN EXTENSION	0	0	1
LINE DRAWING	0	1	0
MATH	0	1	1
BOLD NORMAL ROMAN	1	0	0
BOLD ROMAN EXTENSION	1	0	1
ITALICISED NORMAL ROMAN	1	1	0
ITALICISED NORMAL ROMAN	1	1	1

CHARACTER CODE STRUCTURE

D7	D6	D5	D4	D3	D2	D1	D0
end of line *	7 bit character code **						

* The end of line bit, when set, will cause the remainder of the line to be blanked by the HP 150 video hardware.

** The character sets may be seen by performing "SYSTEM TEST" from the HP 150 Service Keys softkey level.

Finding the Row Pointer Table Origin

In order to find the base address of the row pointer table a call must be made to the firmware "memory manager". The firmware manages the alphanumeric video memory and different versions of the firmware may place the pointer table in different places in RAM. For this reason it is necessary to call the memory manager at run time.

The following assembly language routine will return a pointer which is an offset from the alpha RAM segment at 0D0000H. It is compatible with Microsoft C language function calling conventions.

```
ALPHA_SEG   EQU    0D000H              ;Alpha video RAM segment

DGROUP      GROUP  DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:DGROUP
DATA        ENDS


PGROUP      GROUP      PROG

row_jumps   SEGMENT AT 0000H
            ORG    042FH
mem_mgr     LABEL  FAR
row_jumps   ENDS

PROG        SEGMENT    BYTE PUBLIC 'PROG'
            ASSUME     CS:PGROUP,DS:DGROUP,ES:NOTHING

;-------------------------------------------------------------------------
;  unsigned int vid_tadd()
;
;     /* Returns start address offset of video pointer table.  Segment is
;        'alpha_seg'.  video memory is addressed through a table of pointers
;        to absolute display locations.  This function calls the firmware
;        memory manager and returns the table origin */
;
;
```

```
            PUBLIC vid_tadd
vid_tadd    PROC   NEAR
            PUSH   BP                ;Save BP
            PUSH   DS                ;Save DS
            PUSH   ES                ;Save ES
            MOV    DX,7              ;Call the firmware memory manager with
            PUSH   DX                ; function 7: return segment of row
            CALL   mem_mgr           ; pointer table in BX.
            SUB    BX,ALPHA_SEG      ;Convert to ALPHA_SEG relative address
            MOV    CL,4              ;Multiply by 16
            SHL    BX,CL
            ADD    BX,12H            ;Fixed offset to row table
            MOV    AX,BX
            POP    ES                ;Restore ES
            POP    DS                ;Restore DS
            POP    BP                ;Restore BP
            RET
vid_tadd    ENDP

PROG        ENDS

            END
```

Fetching Row Pointers from the Table

Given the row pointer table origin, the following Microsoft C compatible assembly language function will return the pointer for any alphanumeric video row, zero through 26. Note that the pointer returned from this routine is a word pointer and that it must be multiplied by two to get the correct byte offset for the row origin. This is done in the *qik_char* function which follows. The function uses an external variable *row_tab* which is assumed to contain the result of the *vid_tad* function above. The following C language statement would make that assignment:

```
int  row_tab

row_tab = vid_tadd();
```

Note that the variable *row_tab* is declared external in the following assembly language code and defined in the C module above.

```
            EXTRN  row_tab:WORD      ;Base address of alpha video row pointer
                                     ;  table.  Must be initialized from
                                     ;  'vid_tadd' funtion prior to use.

;-------------------------------------------------------------------------
;  unsigned int row_ptr ( row_num )
;
;     /* Returns start offset address of video row number 'row_num'.
;        Segment is 'alpha_seg'.  Pointer comes from table whose base
;        is 'row_tab' is returned by function 'vid_tadd'. */
;
```

```
rptr_args   STRUC                   ;Stack structure for passed arguments
            DW      ?               ;Callers BP
            DW      ?               ;Return address
row_num     DW      ?               ;'row_num' argument
rptr_args   ENDS

            PUBLIC row_ptr
row_ptr     PROC    NEAR
            PUSH    BP
            MOV     BP,SP
            MOV     BX,[BP].row_num ;'row_num' in BX
            SHL     BX,1            ;4 bytes
            SHL     BX,1            ;per entry
            MOV     AX,DGROUP:row_tab
            ADD     BX,AX           ;Now have offset address of video row ptr
            PUSH    DS
            MOV     DX,ALPHA_SEG
            MOV     DS,DX
            MOV     AL,[BX]         ;Low order row pointer
            MOV     AH,[BX+2]       ;High order row pointer
            POP     DS
            POP     BP
            RET
row_ptr     ENDP
```

Placing a Character in Alpha Memory

The Microsoft C compatible function *qik_char* described below is an example of a routine which uses the row pointer returned by the above function to place a character onto the display in any row and column position.

```
;-------------------------------------------------------------------------
;  qik_char ( row, col, dispword )
;
;   /* Writes dispword (character code and attribute/character set code into
;      alpha video memory at display row number 'row' and column number 'col'.
;      Uses row origin address returned by the 'row_ptr' function.
;
;      Arguments in:
;            'row'       - screen position row number (0 <= row <= 26)
;            'col'       - screen position column number (0 <= col <= 79)
;            'disp_word' - high order byte is attribute/character set byte
;                          and low order byte is character code
;
;      Return values:  none
;
;      Registers preserved:  BP, SP, SI
;
```

```
qchar_args  STRUC                   ;Stack structure for passed argument
            DW      ?               ;Callers BP
            DW      ?               ;Return address
row         DW      ?               ;Screen row number (0 <= row <= 26)
col         DW      ?               ;Screen column number (0 <= col <= 79)
dispword    DW      ?               ;attribute/char set and character bytes
qchar_args  ENDS

            PUBLIC  qik_char
qik_char    PROC    NEAR
            PUSH    BP
            MOV     BP,SP
            MOV     AX,[BP].row     ;Get video RAM address
            PUSH    AX              ;Row on stack for 'row_ptr' call
            CALL    row_ptr
            MOV     DI,AX           ;Lines alpha RAM address word offset
            POP     AX              ;row back off stack
            MOV     AX,[BP].col     ;Column off stack
            ADD     DI,AX           ;Add to row displacement
            SHL     DI,1            ;Two bytes per character
            MOV     AX,[BP].dispword ;Char and enhancement codes from stack
            MOV     BX,DS           ;Save current data segment
            MOV     CX,ALPHA_SEG    ;Initialize alpha RAM segment
            MOV     DS,CX
            MOV     [DI],AX         ;Put char and enhancement to alpha RAM
            MOV     DS,BX           ;Restore original data segment
            POP     BP
            RET
qik_char    ENDP
```

NOTE

It is necessary to initialize the video memory prior to using the above function for random character reads and writes. The firmware sets bit 7 of the character code to blank trailing portions of lines. Any character placed into the line buffer will not be displayed unless all preceeding character codes in the line have their most significant bits cleared (0). The following Microsoft C code uses the *qik_char* routine above to clear all those bits, writing a space with no attributes to display lines 1 through 24.

```
qik_init()

   /* Initializes alpha RAM for random use of the 'qik_char' function.  Writes
      spaces to all character positions on lines zero through 23.  This is
      required to remove any end-of-line markers in the memory.  Should be
      called once prior to using the 'qik_char' function in a random manner  */

{
   int  row;

int  col;

   for ( row = 0; row <= 23; ++row) {
      for (col = 0; col <= 79; ++col) {
         qik_char (row, col, 0x0020 );
      };
   };
}
```

Writing to Lines 25, 26, and 27

The HP 150 uses screen lines 25 and 26 for softkey labels, and line 27 as a status line. It periodically updates softkey labels and the cursor position counters on lines 25 and 26, and the time and status messages on the 27th line from variables it maintains in other places in memory. The previously described character to screen routine *qik_char* is capable of writing to any screen line, including the last three. If that routine is to be used to place text on lines 25 through 27 the firmware should be disabled from overwriting those lines with softkey and status text. The first of the two routines which follow will disable firmware modification of the last three lines of the screen. The second should be called prior to program termination to reenable that processing.

```
PGROUP      GROUP   PROG

firm_jumps  SEGMENT AT 0000H

            ORG     0420H
thp_entry   LABEL   FAR             ;Terminal handler process entry
            ORG     043EH
php_entry   LABEL   FAR             ;Personality handler process

firm_jumps  ENDS

PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME  CS:PGROUP,DS:DGROUP,ES:NOTHING
```

```
;-----------------------------------------------------------------
;  stat_init()
;
;     /* To use the status line on the HP150, one must first disable the
;        portion of the firmware (the status line process) that would
;        otherwise continually update the status line.  After disabling the
;        status line process, a program can write into the video memory
;        representing the status line.
;
;        Arguments:  none
;
;        Returned values:  none
;
;        Registers preserved:  none    */
;
            PUBLIC stat_init
stat_init   PROC   NEAR
            MOV    DX,1
            PUSH   DX
            MOV    DX,2
            PUSH   DX
            CALL   php_entry          ;Get DS register value for the
                                      ; global context area for the
                                      ; terminal personality
            MOV    DS,DX              ;The DS register value is
                                      ; returned in the DX register,
                                      ; move it into the DS register
            MOV    DX,0
            PUSH   DX
            MOV    DX,0               ;Code to disable status line process
            PUSH   DX
            MOV    DX,3               ;Message type to send to status
                                      ; line process
            PUSH   DX
            MOV    DX,[3AH]           ;Using the DS value obtained above
                                      ; get the token for the status
                                      ; line process
            PUSH   DX
            MOV    DX,6
            PUSH   DX
            CALL   thp_entry          ;Call TASK HANDLER PROCESS to
                                      ; disable the status line process
            MOV    DX,0
            PUSH   DX
            CALL   yield              ;The yield routine is described
                                      ; elsewhere
stat_init   ENDP
```

```
;-----------------------------------------------------------------
;  stat_clnu()
;
;     /* This routine renables the firmware status line process causing it
;        to resume updating of softkey labels, cursor position information,
;        time, and status messages on the last three lines of the display.
;
;        Arguments in:  none
;
;        Returned values:  none
;
;        Registers preserved:  none    */
;
            PUBLIC stat_clnu
stat_clnu   PROC   NEAR
            MOV    DX,1
            PUSH   DX
            MOV    DX,2
            PUSH   DX
            CALL   php_entry          ;Get DS register value for the
                                      ; global context area for the
                                      ; terminal personality
            MOV    DS,DX              ;The DS register value is
                                      ; returned in the DX register,
                                      ; move it into the DS register
            MOV    DX,0
            PUSH   DX
            MOV    DX,1               ;Code to enable status line process
            PUSH   DX
            MOV    DX,3               ;Message type to send to status
                                      ; line process
            PUSH   DX
            MOV    DX,[3AH]           ;Using the DS value obtained above
                                      ; get the token for the status
                                      ; line process
            PUSH   DX
            MOV    DX,6
            PUSH   DX
            CALL   thp_entry          ;Call TASK HANDLER PROCESS to
                                      ; enable the status line process
            MOV    DX,0
            PUSH   DX
            CALL   yield              ;The yield routine is described
                                      ; elsewhere
stat_clnu   ENDP

PROG        ENDS

            END
```

GRAPHICS DISPLAY INTERFACING

The display system on the HP 150 consists of two separate "planes", a character based alphanumeric plane and a bit mapped graphics plane 512 dots wide by 390 dots high. The Alphanumeric and Graphics planes overlay each other on the display.

A comprehensive set of high level functions are provided to access the graphics plane. The AGIOS function set (see System Software section) provides functions to perform vector draws, area fills, implement graphics-plane based character sets, etcetera. This is the preferred interface for programmers wishing to access the graphics bit-map. However there are situations where, for performance or ease of software transportation reasons, the programmer may wish to access the graphics bit-map directly. The following information describes how that may be done.

Portion of the 8088 address space is allocated to a 32 kilobyte block of memory, the Graphics RAM (see Memory and I/O Mapping section). Of this 32 kilobyte block, a little over 24 kilobytes forms the graphics bit-map. The remainder is used by the firmware and should not be modified in any way by the programmer. The bit-map is organized as described in the following diagram:

'y' BIT POSITION	'x' BIT POSITION							
	:0 7:	:8 15:	:16 23:	:24 31:	:32 39:	:40	:504 503:	511:
0	C0000H	C0001H	C0002H	C0003H	C0004H			C003FH
1	C0040H	C0041H	C0042H	C0043H	C0044H			C007FH
.								
.								
.								
389	C6140H	C6141H	C6142H	C6143H	C6144H			C617FH

A bit set ("1") will turn on the graphics dot corresponding to that bit position.

An example of an assembly language procedure which accesses the graphics bit map follows. The routine is implemented as a Microsoft C Language compatible function which sets a bit (turns the graphics dot on) given the x and y coordinates.

```
;-------------------------------------------------------------------------
;  bit_set ( x, y )
;
;  /* sets graphics bit map memory location for bit position 'x' on
;     scan line 'y' */
;
;  int  x,y;
;

TRUE        EQU   1                 ;Boolean true for return value
FALSE       EQU   0                 ;Boolean false for return value

dc_args     STRUC                   ;Stack structure for passed parameters
            DW    ?                 ;Callers BP
            DW    ?                 ;Return address
ax_in       DW    ?                 ;'ax'
bx_in       DW    ?                 ;'bx'
cx_in       DW    ?                 ;'cx'
dx_in       DW    ?                 ;'dx'
dc_args     ENDS

ret_args    STRUC                   ;Structure for returned parameters
ax_out      DW    ?                 ;Returned AX word
bx_out      DW    ?                 ;Returned BX word
cx_out      DW    ?                 ;Returned CX word
dx_out      DW    ?                 ;Returned DX word
ret_args    ENDS

DGROUP      GROUP DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:DGROUP

            PUBLIC dc_ret
dc_ret      ret_args <0,0,0,0>      ;Global variable 'dc_ret' declaration

DATA        ENDS
```

```
PGROUP      GROUP   PROG
PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME  CS:PGROUP,DS:DGROUP,ES:NOTHING

bs_args     STRUC                   ;Stack structure for passed arguments
            DW      ?               ;Callers BP
            DW      ?               ;Return address
x           DW      ?               ;x coordinate (0 <= x <= 511)
y           DW      ?               ;y coordinate (0 <= y <= 389)
bs_args     ENDS

            PUBLIC bit_set
bit_set     PROC    NEAR
            PUSH    BP
            MOV     BP,SP
            MOV     BX,[BP].y       ;Get y coordinate (0 <= y <= 389)
            MOV     AX,40H          ;40 bytes per line in bit map
            MUL     BX              ;AX now contains start byte of graphics
                                    ;  raster line y.  Ignore upper word of
                                    ;  result as should be zero.
            MOV     DI,AX
            MOV     DX,[BP].x       ;Get x coordinate (0 <= x <= 511)
            MOV     AX,DX           ;Determine number of whole bytes across
            MOV     CL,3            ;  x coordinate is.
            SHR     AX,CL
            ADD     DI,AX           ;Add to raster line byte offset.
            MOV     CL,DL
            AND     CL,00000111B    ;Determine bit position from x
            MOV     DL,00000001B    ;Create a mask
            ROL     DL,CL
            MOV     CX,0C000H       ;Graphics bit map segment
            MOV     ES,CX
            OR      ES:[DI],DL      ;Set the bit in graphics bit map
            POP     BP
            RET
bit_set     ENDP

data_seg    ENDP

PROG        ENDS

            END
```

As an example of a C program using the above function, consider the following. This program takes 9111A Graphics Template stylus position inputs and scales the x and y coordinates down to the HP 150 graphics bit-map matrix size. The corresponding dot in the graphics bit-map is then turned on. In this way, the graphics template stylus acts as an "electronic pencil" for the HP 150 graphics display. Note that the stylus status check inhibits bit-map modification unless the stylus is depressed.

The following main calling routine requires the following support code which is described elsewhere in this section:

- *doscall* function, see "MS-DOS System Function Calls from C Language".

- All routines (except *main*), structures, variables and constants from "HPIB Interface Example (911A Graphics Tablet)".

```
/*****************************************************************************/
main()
{
   int x, y, status;

   if ( tab_init() == 1 ) {                  /* initialize graphics tablet */
      printf ( "%c*dA\n\r", ESC );           /* clear graphics memory */
      printf ( "%c*dC\n\r", ESC );           /* turn on graphics memory */
      while ( kbhit() == 0 ) {
         get_x_y_status (&x, &y, &status, tab_handle);  /* get data, */
                                             /* if stylus within bounds and
                                                is depressed, put dot on display*/
         if ( (x <= 0x2de9) && (y <= 0x216e) &&
              ( (status == 0x519) || (status == 0x719) ) ) {
            x = x/23;                        /* convert stylus position to */
            y = 389 - (y/22);                /* display bit map position   */
            bit_set ( x, y );                /* turn on graphics memory bit */
         };
      };
   };

   xenixclose (tab_handle);                  /* close the tablet */
}
```

KEYBOARD INTERFACING

The HP 150 keyboard includes many special function keys such as "Clear display". These keys are processed locally by the HP 150 firmware rather than being sent to an application requesting its information through the console input function and its associated high-level language inplementations (for example BASIC "INPUT"). Applications may obtain additional control over the HP 150 keyboard by placing the HP 150 keyboard into what is termed "keycode mode". This mode allows such keys as *Clear display* to be passed to the application program rather than resulting in the display being cleared.

Keycode Mode

Through the use of keycode mode, applications can detect activation of almost any HP 150 keyboard key as well as the state of the SHIFT, EXTEND CHAR, and CTRL keys in conjunction with other keys. Keycode mode also allows programs to receive touchscreen input and distinguish it from keyboard input.

The following routines are examples showing how the HP 150 might be put into keycode mode. The AGIOS calls are more thoroughly described in the HP 150 Programmer's Reference Manual. The first function uses the AGIOS Keycode ON/OFF function to read and save the keycode mode (on or off). It could be used so that the following function (*km_rest*) could restore the entry mode prior to program termination. The other two functions set keycode mode on and off respectively. Note that all functions are written in the C language and use the *agios* and *data_seg* functions described elsewhere in this section.

```
#define ON  1
#define OFF 0

struct   agios_ko_tem   {          /* AGIOS Keycode Mode ON/OFF parameter temp*/
                        unsigned int funct_num;
                        char         mode;
                        };
#define KO_FLEN         sizeof(struct agios_ko_tem)
#define KO_FNUM         43         /* Keycode mode ON/OFF function # */

struct   agios_ks_tem   {          /* AGIOS Key Status parameter template */
                        unsigned int funct_num;
                        unsigned int buf_offset;
                        unsigned int buf_segment;
                        };
#define KS_FLEN         sizeof(struct agios_ks_tem)
#define KS_FNUM         44         /* Read keycode mode status function # */

struct  agios_ko_tem  agios_ko_par  = {KO_FNUM, 0}; /* AGIOS set keycode
                                                       mode command buffer */
struct  agios_ks_tem  agios_ks_par;                 /* AGIOS read keycode
                                                       mode command buffer */
unsigned int entry_kcm;
int dseg;
```

```
km_get()
{
   dseg = data_seg();
   agios_ks_par.funct_num = KS_FNUM;    /* Read and save keycode mode status  */
   agios_ks_par.buf_offset = &entry_kcm;
   agios_ks_par.buf_segment = dseg;
   agios(&agios_ks_par, KS_FLEN);
}

km_rest()
{
   agios_ko_par.mode = entry_kcm;       /* Restore keycode mode state         */
   agios(&agios_ko_par, KO_FLEN);
}

km_on()
{
   agios_ko_par.mode = ON;              /* Turn on keycode reporting mode     */
   agios(&agios_ko_par, KO_FLEN);
}

km_off()
{
   agios_ko_par.mode = OFF;             /* Turn off keycode reporting mode    */
   agios(&agios_ko_par, KO_FLEN);
}
```

Console RAW/COOKED Mode

It is advisable that MS-DOS be placed in RAW mode when doing keycode mode inputs from the keyboard, otherwise MS-DOS will process the key information looking for special control sequences such as Control-C (terminate program) and Control-P (toggle printer logging). The following routines show how the RAW/COOKED mode for the console may be manipulated. They are written in the C language and use the *doscall* function described earlier in this section. The first two functions are intended for initialization and cleanup respectively. They can be used to save the RAW/COOKED mode as at program entry, and restore it prior to termination. See the Programmer's Reference Manual for more information on the MS-DOS I/O Control for Devices System Function Call.

```
#define RAW_MASK 0x20      /* MSDOS ioctrl - raw mode bit mask          */
#define IOC_GET  0x4400    /* MS-DOS ioctrl - get device information */
#define IOC_SET  0x4401    /* MS-DOS ioctrl - set device information */
#define STDIN    0         /* standard input MS-DOS file handle */

char entry_ioc;
```

```
raw_get()
{
   doscall ( IOC_GET, STDIN );          /* Read and save raw/cooked mode */
   entry_ioc = ( dc_ret.dl ) & 0x00FF;  /* MSB of entry_ioc must be zero
                                           before it is used to set mode */
}

raw_rest()
{
   doscall(IOC_SET, STDIN, 0, entry_ioc);   /* Restore raw/cooked mode state */
}

raw_on()
{
   doscall(IOC_SET, STDIN, 0, (entry_ioc | RAW_MASK)); /* Set raw mode on    */
}

raw_off()
{
   doscall(IOC_SET, STDIN, 0, (entry_ioc & ~RAW_MASK)); /* Turn off raw mode */
}
```

Key Characteristics

While in keycode mode, the characteristics of each key may be set individually. Specifically each key may be:

- Processed normally (for example *Clear display* clears the display)
- Intercepted and passed to the application for processing
- Ignored
- Any of the above accompanied by a bell beep

The following C language functions turn the interception mode on and off for all keys on the keyboard. The programmer could modify these routines to selectively define individual keys with any of the above characteristics. Note that it is not possible to read and save the key characteristics prior to changing them so that they could be restored prior to program termination. See the HP 150 Programmer's Reference Manual for more information on the AGIOS Define Key Characteristics function.

```
struct   agios_dk_tem   {          /* AGIOS Define Key Characteristics */
                        unsigned int funct_num;
                        unsigned int characteristics;
                        unsigned int keycode;
                        };
#define DK_FLEN         sizeof(struct agios_dk_tem)
#define DK_FNUM         40         /* Define Key characteristics function # */

struct  agios_dk_tem  agios_dk_par = {DK_FNUM, 0x0002, 0x00FE};  /* AGIOS
                                     define key characteristics command buffer */
```

```
int_on()
{
   agios_dk_par.characteristics = 0x0002; /* Turn on key intercept mode      */
   agios(&agios_dk_par, DK_FLEN);         /*    for all keys                */
}

int_off()
{
   agios_dk_par.characteristics = 0x0000; /* Turn off key intercept mode     */
   agios(&agios_dk_par, DK_FLEN);
}
```

Sample Keyboard Driver

The following program and its support function 'key_hit' demonstrate how one might use keycode mode. The main calling routine initializes the HP 150 by saving the state of RAW/COOKED and keycode modes prior to setting them on. It then loops printing keyboard statistics for keys pressed and terminates following depression of the "Stop" key. Prior to termination, RAW/COOKED and keycode modes are restored as at entry.

```
#define KBD_STATUS 0x4406    /* MS-DOS ioctrl - get input status */
#define READ_FC    0x3F00    /* MS-DOS read file/device function number */
#define TRUE       1
#define FALSE      0

struct  key_buf_tem {   /* keycode buffer template */
                    char flags;
                    char id;
                    char keycode;
                    char unused;
                    };

struct  key_buf_tem  key_buf; /* Holds a single entry read from
                                 the keyboard buffer while in keycode mode */
```

```
main()
{
   raw_get();     /* remember current console RAW/COOKED mode */
   km_get();      /* remember current keycode mode */

   raw_on();      /* turn on console RAW mode */
   int_on();      /* turn on key intercept mode for all keys */
   km_on();       /* turn on keycode mode */

   do {
      if ( key_hit() ) {
         printf ( "Data from keyboard: FLAGS = %x;  ID = %x;  KEYCODE = %x\r\n"
                  , key_buf.flags, key_buf.id, key_buf.keycode );
      };
   } while ( key_buf.keycode != 0x58 );
   km_rest();     /* restore keycode mode as at entry */
   int_off();     /* turn intercept mode off again */
   raw_rest();    /* restore raw mode as at entry */
}


key_hit()

   /* Checks for data in the HP-150 keyboard type ahead buffer.  This routine
      checks to see if there are any keys in the keyboard type ahead buffer.
      If so the data for the first key is read from the buffer and placed in
      a structure variable.

      Arguments in:  None.
      Return value:
         TRUE if a key was read from the buffer
         FALSE if nothing was read from the buffer */

{
   doscall(KBD_STATUS, STDIN, 0, 0);          /* If there are any keys in the  */
   if ( dc_ret.al ) {                         /*    keyboard type ahead buffer */
      doscall(READ_FC, STDIN, 4, &key_buf);   /*    then read the data bytes   */
      return(TRUE);                           /*    for the first key          */
   }
   return(FALSE);
}
```

Flushing the Keyboard Buffer

The following routine flushes the keyboard type-ahead buffer by reading all the keys from it until none are left. Note that the keyboard type ahead buffer is a firmware buffer, not an MS-DOS buffer. The only way to flush the firmware buffer is to read keys from the MS-DOS device until the buffer is empty. After reading a key from the MS-DOS device the MS-DOS task must yield to the firmware to allow it to remove another key from its buffer and pass it to the MS-DOS task. An assembly language routine which will yield control to the firmware follows the *flush_kbuf* routine.

```
flush_kbuf()

   /* Flush HP-150 keyboard type ahead buffer */

{
   struct key_buf_tem trash_can;    /* temporary buffer to deposit keys */

   doscall ( KBD_STATUS, STDIN, 0, 0 );      /* Read everthing in the type */
   while ( dc_ret.al ) {                     /* ahead buffer and ignore it */
      doscall ( READ_FC, STDIN, 4, &trash_can );
      yield ( 1 );                           /* Yield to allow the firmware to*/
      doscall ( KBD_STATUS, STDIN, 0, 0 );   /* Put any pending keys in the */
   }                                         /* MSDOS key buffer */
}


YIELD_FCN   EQU   009H                ;Task handler yield function code

DGROUP      GROUP  DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:DGROUP

THP_ENTRY   LABEL   DWORD                     ;Task handler entry point
THP_OSET    DW      00420H                    ;Offset
THP_SEG     DW      00000H                    ;Segment

DATA        ENDS

PGROUP      GROUP  PROG
PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME CS:PGROUP, DS:DGROUP, ES:NOTHING

;-------------------------------------------------------------------------------
;  yield ( time )                  /* Yield control to firmware */
;     int time;                    /* Time parameter to firmware */
;
;     Calls the firmware task handler with a function code indicating that
;     any pending firware tasks should be allowed to process for a specified
;     period of time.
;
;     Arguments in:
;        time - minimum number of milliseconds that the firmware will have
;               control up to a maximum value determined by the firmware.  A
;               value of zero will simply cause the MSDOS task to be placed
;               on the end of the firmware ready task list.
;     Return value:  None.
;
;     Registers preserved:  BP, ES
;
```

```
YIELD_ARGS  STRUC                       ;Stack structure for passed args.
            DW      ?                   ;Caller's BP
            DW      ?                   ;Caller's ES
            DW      ?                   ;Return address
YIELD_TIME  DW      ?                   ;Minimum milliseconds for yield
YIELD_ARGS  ENDS

            PUBLIC YIELD
YIELD       PROC    NEAR                ;Yield to firmware tasks
            PUSH    ES                  ;Save callers ES
            PUSH    BP                  ;Save callers BP
            MOV     AX,[BP].YIELD_TIME  ;Yield time parameter
            PUSH    AX
            MOV     AX,YIELD_FCN        ;Yield function code
            PUSH    AX
            CALL    DGROUP:THP_ENTRY    ;Call firmware task handler
            POP     BP                  ;Restore callers BP
            POP     ES                  ;Restore callers ES
            RET                         ;Return to caller
YIELD       ENDP

PROG        ENDS

            END
```

Keycode Tables

		USASCII KEYCODES					
Key #	Physical Key addr	Normal	Control	Shift	Extended Character 8-bit yes	8-bit no	Special flag
001	057	054	054	FFF	054	054	1
002	03A	058	FFF	05A	058	058	1
003	03D	000	000	000	000	000	1
004	03E	001	001	001	001	001	1
005	03F	002	002	002	002	002	1
006	038	003	003	003	003	003	1
007	008	053	FFF	059	053	053	1
008	00F	051	FFF	FFF	051	051	1
009	00E	004	004	004	004	004	1
010	00D	005	005	005	005	005	1
011	00C	006	006	006	006	006	1
012	00B	007	007	007	007	007	1
013	00A	043	043	049	043	043	1
014	009	042	069	048	042	042	1
015	03C	060	000	07E	0FB	060	0
016	02C	031	FFF	021	0B8	031	0
017	051	032	FFF	040	040	032	0
018	044	033	FFF	023	023	033	0
019	045	034	FFF	024	0F7	034	0
020	046	035	FFF	025	0F8	035	0
021	047	036	FFF	05E	05E	036	0
022	040	037	FFF	026	05C	037	0
023	010	038	FFF	02A	05B	038	0
024	017	039	FFF	028	05D	039	0
025	016	030	FFF	029	0B9	030	0
026	015	02D	02D	05F	0F6	02D	0
027	014	03D	03D	02B	0FE	03D	0
028	013	027	027	027	027	027	1
029	012	044	044	044	044	044	1
030	011	045	045	045	045	045	1
031	049	024	024	026	024	024	1
032	04A	071	011	051	FFF	071	0
033	04B	077	017	057	07E	077	0
034	04C	065	005	045	0D7	065	0
035	04D	072	012	052	FFF	072	0
036	04E	074	014	054	FFF	074	0
037	04F	079	019	059	FFF	079	0
038	048	075	015	055	FFF	075	0
039	018	069	009	049	FFF	069	0
040	01F	06F	00F	04F	0D6	06F	0

		USASCII KEYCODES					
Key #	Physical Key addr	Normal	Control	Shift	Extended Character 8-bit yes	8-bit no	Special flag
041	01E	070	010	050	0F1	070	0
042	01D	05B	01B	07B	0B3	05B	0
043	01C	05D	01D	07D	07C	05D	0
044	01B	05C	01C	07C	FFF	05C	0
045	01A	040	06B	046	040	040	1
046	019	041	06D	047	041	041	1
047	043	056	056	056	056	056	1
048	03B	FFF	FFF	FFF	FFF	FFF	0
049	053	061	001	041	0D4	061	0
050	054	073	013	053	0DE	073	0
051	055	064	004	044	0E4	064	0
052	056	066	006	046	0BE	066	0
053	039	067	007	047	0BA	067	0
054	050	068	008	048	0BC	068	0
055	020	06A	00A	04A	024	06A	0
056	027	06B	00B	04B	0BF	06B	0
057	026	06C	00C	04C	0BB	06C	0
058	025	03B	03B	03A	0AF	03B	0
059	024	027	027	022	060	027	0
060	023	025	025	025	025	025	1
061	022	02C	06E	02D	02C	02C	1
062	021	02F	02F	02F	02F	02F	1
063	071	055	055	057	055	055	1
064	041	FFF	FFF	FFF	FFF	FFF	0
065	073	07A	01A	05A	FFF	07A	0
066	074	078	018	058	0EC	078	0
067	075	063	003	043	0B5	063	0
068	076	076	016	056	0BD	076	0
069	077	062	002	042	0FC	062	0
070	070	06E	00E	04E	0F9	06E	0
071	028	06D	00D	04D	0FA	06D	0
072	02F	02C	02C	03C	03C	02C	0
073	02E	02E	02E	03E	03E	02E	0
074	02D	02F	02F	03F	05F	02F	0
075	042	FFF	FFF	FFF	FFF	FFF	0
076	02B	052	052	052	052	052	1
077	02A	020	065	029	020	020	1
078	029	02E	06C	02E	02E	02E	1
079	072	050	05C	05B	050	050	1
080	036	FFF	FFF	FFF	FFF	FFF	0

		USASCII KEYCODES					
Key #	Physical Key addr	Normal	Control	Shift	Extended Character 8-bit yes	8-bit no	Special flag
081	037	020	020	020	020	020	0
082	035	FFF	FFF	FFF	FFF	FFF	0
083	033	023	061	02A	023	023	1
084	032	021	062	028	021	021	1
085	031	022	063	02B	022	022	1
086	06F	008	008	008	008	008	1
087	06D	009	009	009	009	009	1
088	06B	00A	00A	00A	00A	00A	1
089	069	00B	00B	00B	00B	00B	1
090	067	071	071	071	071	071	1
091	065	073	073	073	073	073	1
092	063	072	072	072	072	072	1
093	061	070	06A	070	070	070	1
094	05F	037	037	037	037	037	1
095	05D	038	038	038	038	038	1
096	05B	039	039	039	039	039	1
097	059	075	075	075	075	075	1
098	05E	034	034	034	034	034	1
099	05C	035	035	035	035	035	1
100	05A	036	036	036	036	036	1
101	058	074	074	074	074	074	1
102	066	031	031	031	031	031	1
103	064	032	032	032	032	032	1
104	062	033	033	033	033	033	1
105	06E	030	030	030	030	030	1
106	06A	077	077	077	077	077	1
107	060	076	076	076	076	076	1

NOTE

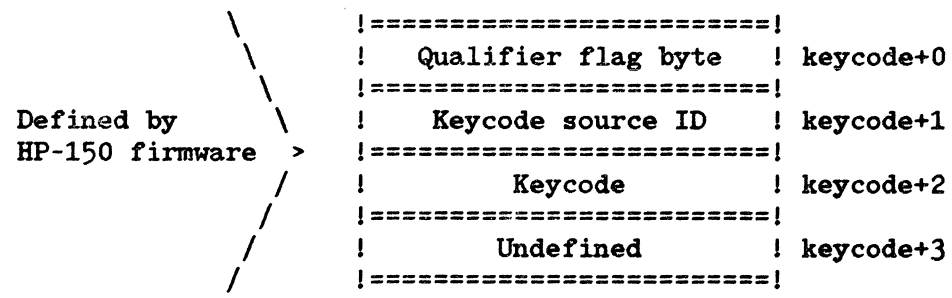
1. The keyboard key numbers are specified in decimal, all other entries are specified in hexadecimal.

2. A keycode entry of 'FFF' means that no keycode is returned for that key sequence.

3. The keycodes were obtained using Rev. B of the firmware ROM's:
   1818-3321, 1818-3322, 1818-3323, 1818-3324, 1818-3325
   Keycode reporting differences in Rev. A of the firmware ROM's:
   CTRL <key 072> does not report a keycode
   CTRL <key 073> does not report a keycode
   CTRL <key 097> reports keycode 05C

4. The keycodes were recorded with the keyboard state set as follows:
   Keycode mode on.
   Console device in raw mode.
   Intercept mode set for all keys.
   Control-C checking off.
   Caps mode off (keyboard key function).
   Caps lock off (configuration menu function).
   Language USASCII (configuration menu function).
   Application softkey labels displayed.

5. Every keycode returned to an application is made up of four bytes:



   Qualifier byte format:

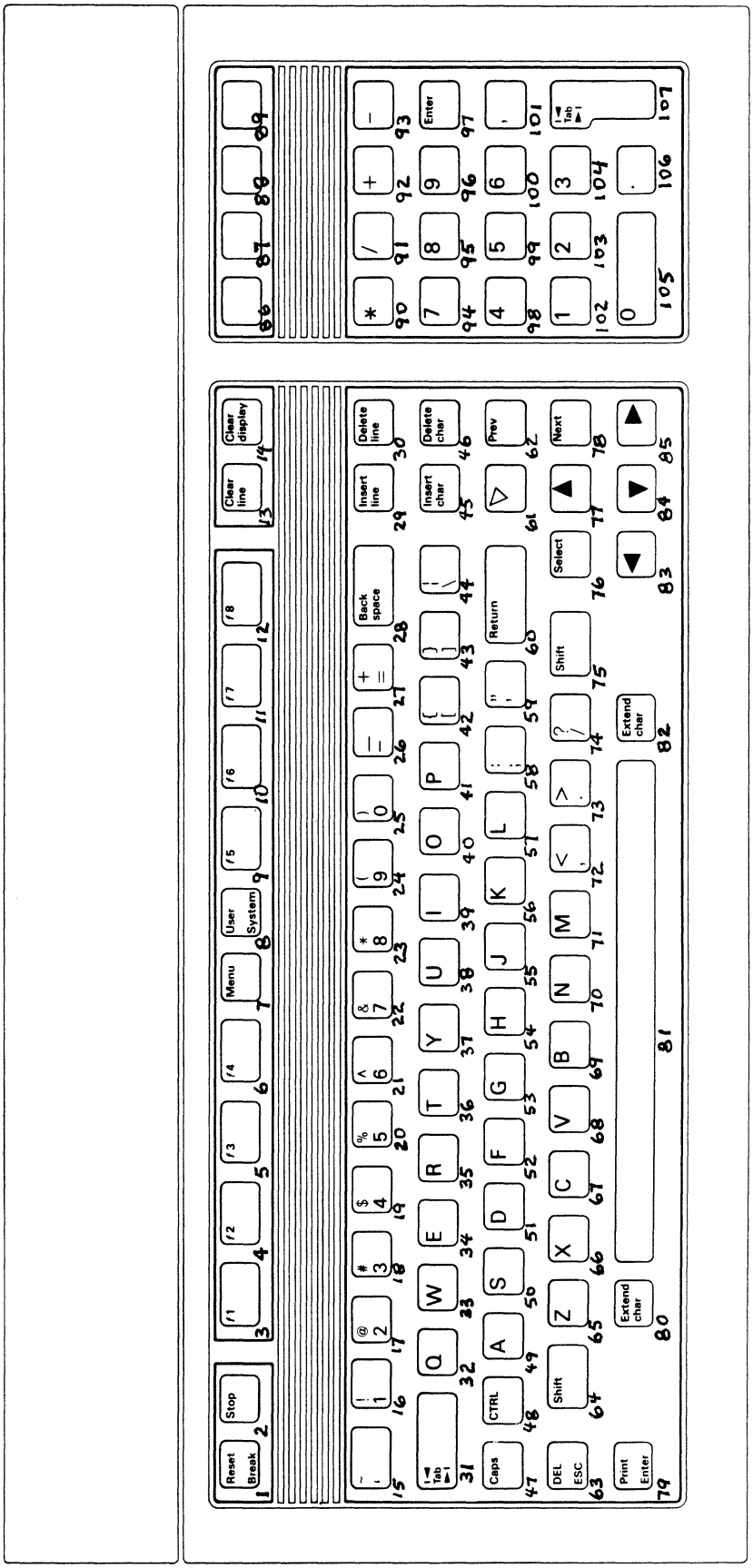
SP	XX	LE	RE	CT	LS	RS	RP

   SP - Special key
   XX - Reserved
   LE - Left extended character key down
   RE - Right extended character key down
   CT - Control key down
   LS - Left shift key down
   RS - Right shift key down
   RP - Repeating key

   Keycode source ID byte:
   0C0H - Keyboard data
   080H - Touchscreen data

6. Keycodes vary with the keyboard (language) field of the terminal configuration menu.

7. CTRL-numeric keys on main keyboard section (not numeric) report keycodes in an erratic manner. These key sequences should not be used for this reason.



Keyboard Key Position Numbers

PROGRAMMING DATA COMMUNICATIONS

The HP 150 personal computer has comprehensive data communications facilities which are accessible to the programmer. There are two data communications ports, both of which are capable of running in synchronous or asynchronous mode. Only asynchronous mode is supported by the system firmware and software. If you wish to program the HP 150 to use synchronous mode it will be necarrary to write your own hardware level I/O, configuration, and interrupt drivers. For persons wishing to do such things, see the "Data Comm at the Hardware I/O Level" discussion later in this section.

The following information should make your task of interfacing to the HP 150's powerful data communications capabilities relatively simple, especially for asynchronous operation. The discussion covers logical (COM1 and COM2) to physical (Port1 and Port2) configuration, input and output of data through these devices, programmatic reconfiguration of the port asynchronous operating parameters, and port hardware interfacing. Included is an example of a simple terminal emulator application program.

Data Comm Using COM1 and COM2 Logical Devices

Input/Output to the COM1 and COM2 devices can be performed using the MS-DOS "Read from a File/Device" and "Write to a File/Device" System Function Calls. The device in question must be "opened" prior to accessing it in this manner, and should be "closed" when access is complete, usually just prior to application program termination. It may be necessary to have the user assign COM1 and/or COM2 to one of the physical serial I/O ports on the HP 150 prior to running your application.

NOTE

Serial communications I/O through the MS-DOS COM devices is somewhat slow because of operating system and firmware overhead. In practice throughput rates of around 2400 baud (half duplex) can be attained. For higher throughput it may be necessary to use the MS-DOS I/O Control for Devices System Function Call or deal directly with the hardware. See "Data Comm Control Functions" later in this section.

ASSIGNING COM1 AND COM2 TO PHYSICAL PORTS

COM1 and COM2 are recognized as device names by MS-DOS. They may be mapped (assigned) to either of the physical hardware ports, "Port1" and "Port2". There is an RS232 connector for Port1 and an RS232/422 connector for Port2 at the rear of the HP 150. Both these ports are standard on the HP 150. The mapping between COM1/COM2 and Port1/Port2 is stored in non-volatile memory on the HP 150. Currently the only trivial way to change this assignment is to run the "DEVCONFG" utility program shipped with the HP 150 operating system. The default mapping has COM1 assigned to Port1 and COM2 assigned to Port2. This means that MS-DOS references to COM1 cause Port1 hardware to be accessed, and references to COM2 cause Port2 hardware to be accessed. The assignment of Port1

and Port2 as the "Remote/Serial" devices in the DEVCONFG utility has no meaning here and will not alter the logical to physical device mapping.

OPENING THE COM DEVICES

The MS-DOS "Open a File" System Function Call (3DH) provides a "handle" or device identifier used for input and output of data and control information to the device. The function call is made passing a pointer to a character string being the device name, COM1 or COM2 in this case. The character string must be terminated by a null byte. The following is a C language example of a call to open the COM1 device. It uses the *doscall* function previously described.

```
#define OPEN_RW  0x3D02   /* MS-DOS Open File/Device System Function */
#define DUMMY    0        /* dummy 'doscall' argument */

struct dc_ret_tem  {     /* 'doscall' return parameter structure */
                   char al;
                   char ah;
                   char bl;
                   char bh;
                   char cl;
                   char ch;
                   char dl;
                   char dh;
                   };

extern  struct  dc_ret_tem  dc_ret;

char com1_name[5] = { "COM1" };
char comhandle;

{
   com1_name[4] = 0;                 /* terminate device name string with NULL */
   doscall ( OPEN_RW, DUMMY, DUMMY, &com1_name[0] );    /* open COM1: device */
   comhandle = dc_ret.al;
}
```

The above code associates the variable *comhandle* with a handle used for all future references to the COM1 device while it remains open.

INPUT FROM THE COM DEVICES

As mentioned, data input from the serial communications devices is most easily performed using the MS-DOS "Read from a File/Device" System Function Call (3FH). This function allows single or multiple character input from the device. The Read from a File/Device function will wait on a character if none exists when called. The firmware level drivers for Port1 and Port2 buffer incoming and outgoing data. If the port incoming buffer is empty when the read function is called, the function will not return with a character until a byte is received

at the serial interface for the port. Otherwise the function will return a character from the buffer and adjust the buffer accordingly.

The MS-DOS I/O Control for Devices System Function Call can be used to check the Port1 and Port2 incoming buffer status. The function is called using the "get input status" request as in the following C language example.

```
#define IOC_INSTAT 0x4406          /* MS-DOS IOCTL Function - get input status */

{
   doscall ( IOC_INSTAT, comhandle );              /* check device input stat  */
   if ( dc_ret.al == 0xFF )
      /* buffer contains data */;
   else
      /* buffer is empty */;
```

The above I/O Control for Devices function with the get input status request would normally be called prior to reading a character from the device. Note that this may not be a valid function call for devices other than COM1 and COM2. The following is a C language example of a single character input from a open device with the handle *comhandle*. It uses the *doscall* function previously described. The input character is placed in variable *inchar*.

```
#define READ     0x3F00      /* MS-DOS Read From File/Device System Function */

char inchar;

{
   doscall ( READ, comhandle, 1, &inchar );          /* char in from device */
}
```

OUTPUT TO THE COM DEVICES

The MS-DOS "Write to a File/Device" System Function Call allows single or multiple character output to the serial communication devices among others. Once again, it is advisable to check status prior to making the call in case the device outgoing buffer is full. The following example shows how this may be done in the C language.

```
#define IOC_OUTSTAT 0x4407         /* MS-DOS IOCTL Function - get output stat */

{
   doscall ( IOC_OUTSTAT, comhandle );             /* check device output stat */
   if ( dc_ret.al == 0xFF )
      /* buffer can accept data */;
   else
      /* buffer is full */;
```

Given that ready is returned by the check output status call above, the MS-DOS "Write to a File/Device" System Function Call is used in the following example to send the character variable *outchar*.

```
#define WRITE       0x4000    /* MS-DOS Write To File/Device System Function */

char outchar;

{
   doscall ( WRITE, comhandle, 1, &outchar );        /* write char. to device */
}
```

CLOSING THE COM DEVICES

An application which has opened devices or files should close them prior to termination. If either of the MS-DOS COM devices are opened, as COM1 is above, they should be closed using the MS-DOS "Close a File Handle" System Function Call (3EH). The following example shows how that device may be closed in C language. *comhandle* and *com1_name* variables are as defined in the device open example as previously described.

```
{
   doscall ( CLOSE, comhandle );                    /* close the COM1: device */
}
```

COM DEVICE I/O EXAMPLE - A TERMINAL EMULATOR

The following example implements a simple terminal emulator for the HP 150. It will use the COM1 device and sends all escape and control sequences as well as special keyboard keys to the firmware for processing in a manner consistent with "terminal" mode on the HP 150. Pressing ! on the keyboard terminates the application. To use this application on a HP3000 computer, the Port to which COM1 is assigned should be configured as for HP3000 Point-to-Point (system defaults).

```
#define TRUE      1
#define FALSE     0
#define RAW_MASK 0x20      /* MSDOS ioctrl - raw mode bit mask        */
#define OPEN_RW     0x3D02 /* MS-DOS Open File/Device System Function */
#define CLOSE       0x3E00 /* MS-DOS Close File/Device System Function */
#define READ        0x3F00 /* MS-DOS Read From File/Device System Function */
#define WRITE       0x4000 /* MS-DOS Write To File/Device System Function */
#define IOC_GET     0x4400 /* MS-DOS IOCTL Function - get device information */
#define IOC_SET     0x4401 /* MS-DOS IOCTL Function - set device information */
#define IOC_INSTAT  0x4406 /* MS-DOS IOCTL Function - get input status */
#define IOC_OUTSTAT 0x4407 /* MS-DOS IOCTL Function - get input status */
#define STDOUT    1        /* standard output MS-DOS file handle (console) */
#define DUMMY     0        /* dummy 'doscall' argument */

struct dc_ret_tem  {     /* 'doscall' return parameter structure */
                   char al;
                   char ah;
                   char bl;
                   char bh;
                   char cl;
                   char ch;
                   char dl;
                   char dh;
                   };

char dcent_ioc;
char comhandle;
char outchar;
char inchar;
char terminate;
char com1_name[5] = { "COM1" };

extern  struct  dc_ret_tem  dc_ret;

main()
{
   com1_name[4] = 0;                /* terminate device name string with NULL */
   doscall ( OPEN_RW, DUMMY, DUMMY, &com1_name[0] );    /* open COM1: device */
   comhandle = dc_ret.al;
   doscall ( IOC_GET, comhandle );  /* Read and save console raw/cooked mode */
   dcent_ioc = ( dc_ret.dl );
   doscall(IOC_SET,comhandle,0,(dcent_ioc | RAW_MASK));   /* Set raw mode on */
   terminate = FALSE;
```

```
    do {

        if ( kbhit() != 0x00 ) {
            outchar = getch();                          /* get character from kbd */
            if ( outchar == '!' )                      /* end program when ! pressed */
                terminate = TRUE;
            doscall ( WRITE, comhandle, 1, &outchar );  /* write char. to COM1: */
        };
        doscall ( IOC_STAT, comhandle );               /* check COM1: input status */
        if ( dc_ret.al == 0xFF ) {
            doscall ( READ, comhandle, 1, &inchar );       /* char in from COM1: */
            do {                                       /* wait for COM1 output ready */
                doscall ( IOC_OUTSTAT, comhandle );
            } while ( dc_ret.al != 0xFF );
            doscall ( WRITE, STDOUT, 1, &inchar );  /* character out to console */
        };
    } while ( !terminate );
    doscall(IOC_SET,comhandle,0,dcent_ioc);  /* Restore raw/cooked mode state */
    doscall ( CLOSE, comhandle );                      /* close the COM1: device */
}
```

Programmatic Configuration of Data Comm

The current configuration of each data communications port may be determined by making a call to the HP 150 data communications driver firmware. The calls are made by referencing a particular *physical* port - that is, Port1 or Port2. Note that either of these ports are configurable as MS-DOS devices COM1 or COM2. Currently there is no trivial way of determining the mapping in a programmatic manner. This information is stored in non-volatile CMOS memory and is managed by the DEVCONFG utility. The location of the mapping data which the DEVCONFG utility manages is dynamic, that is it varies from version to version of the utility. Unlike the firmware which may be called to return configuration information which it manages, the DEVCONFG utility has no such facility. Accessing the CMOS memory directly to manipulate or read the Port1/Port2 to COM1/COM2 mapping data will introduce DEVCONFG version dependencies. This should be avoided at all costs. Stated in simple terms, if your application must reconfigure either of the serial ports using the methodology to be described, your application will require the user to have fixed Port1/Port2 to COM1/COM2 mapping.

READING THE CURRENT CONFIGURATION

The firmware call to be described will read configuration menu items into a data structure defined by the caller. The structure has a fixed format and is 7 bytes long. Three additional header bytes and one additional trailer word are defined as part of the structure. These bytes are used for firmware use during the read configuration described here and the verify and reconfigure datacomm functions to be described later.

The configuration data structure elements (ignoring the bytes used by the firmware) are defined as follows:

misc_1:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
1 Stop Bit	CS(CB) Xmit	SRR Invert	RR(CF) Recv	SRR Xmit	SR(CH)	Chk Parity	ENQ ACK
0 2Stop	0 No	0 No	0 No	0 No	0 Lo	0 No	0 No
1 1Stop	1 Yes	1 Yes	1 Yes	1 Yes	1 Hi	1 Yes	1 Yes

misc_2:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
TR(CD)	DM(CC) Xmit					7 Data Bits	Xmit Pace
0 Lo	0 No					0 8Data	0 None
1 Hi	1 Yes					1 7Data	1 Xon/Xoff

baud_rate:

02H	110 Baud
04H	150 Baud
05H	300 Baud
06H	600 Baud
07H	1200 Baud
0AH	2400 Baud
0CH	4800 Baud
0EH	9600 Baud
0FH	19200 Baud

parity:

0	Zeros
1	Ones
2	Even
3	Odd
4	None

clock:

0	INT CLOCK
1	EXT X1 CLOCK
2	EXT X16 CLOCK

asterisk:

0	No Asterisk
1	DM Asterisk
2	RR Asterisk
3	Line Asterisk
4	CS Asterisk

recv_pace:

0	No Recv Pace
1	Xon/Xoff Recv Pace
2	TR(CD) Recv Pace

The above structure elements plus some firmware variables are represented in an assembly language structure as follows:

```
config_data STRUC                   ;Structure defining config data area
predef_val  DB    ?                 ;Used by firmware for config calls
barrel_idx  DB    ?                 ;Used by firmware for config calls
port        DB    ?                 ;Used by firmware for config calls

misc_1      DB    ?                 ;Miscellaneous configuration bits
misc_2      DB    ?                 ;Miscellaneous configuration bits
baud_rate   DB    ?                 ;Baud rate
parity      DB    ?                 ;Parity and checking
clock       DB    ?                 ;Clock source
asterisk    DB    ?                 ;Control line asterisk
recv_pace   DB    ?                 ;Receive pacing

dev_token   DW    ?                 ;Used by firmware for config calls
config_data ENDS
```

A Microsoft C language compatible call is now described. The function is passed two parameters:

- The port number (1 = Port1, 2 = Port2)

- A pointer to a data structure defined as above

The function will fill in the structure provided by the caller with the configuration for the appropriate port. Note that the configuration information will reflect that as it currently exists in non-volatile memory, as is seen in the configuration memu.

```
TRUE                EQU  1
FALSE               EQU  0
SUCCESS             EQU  0
DHP_RELEASE_DEVICE  EQU  3
DHP_GDEV            EQU  1
DHP_GDTOK           EQU  14
VERIFY_CODE         EQU  14
CONFIG_CODE         EQU  1
START_DRIVER_CODE   EQU  13


DGROUP      GROUP DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:DGROUP

hpibdev     DB    'HPIBDEV',0       ;HPIB device name string
            PUBLIC fw_es
fw_es       DW    ?                 ;Firmware ES saved here

DATA        ENDS
```

```
PGROUP      GROUP PROG
            ASSUME CS:firm_jumps,DS:NOTHING
firm_jumps  SEGMENT BYTE AT 0000H
            ORG   0425H
dhp_entry   LABEL FAR
            ORG   042AH
dhp_driver  LABEL FAR
            ORG   045CH
cnp_entry   LABEL FAR
firm_jumps  ENDS

PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME CS:PGROUP,DS:DGROUP,ES:NOTHING

;-------------------------------------------------------------------------
;  char get_conf ( port_num, scratch )
;
;     /* Reads current datacomm port number 'port_num' configuration
;        information from CMOS (configuration) memory into the scratch
;        structure at DS:'scratch'.  The configuration data area at 'scratch'
;        will have all the values needed to specify the datacomm
;        configuration.  These can be modified and passed to the firmware
;        datacomm driver to set a new configuration.  Returns TRUE if
;        successful in performing the operation, false otherwise.
;
getcf_args  STRUC                   ;Stack structure for passed arguments
            DW    ?                 ;Callers BP
            DW    ?                 ;Callers return address
port_num    DW    ?                 ;Port number (1 = Port1, 2 = Port2)
scratch     DW    ?                 ;'scratch' argument
getcf_args  ENDS
```

```
            PUBLIC get_conf
get_conf    PROC  NEAR
            PUSH  BP
            MOV   BP,SP
            MOV   BX,[BP].port_num
            MOV   DI,[BP].scratch     ;'scratch' argument from stack
            POP   BP
            MOV   ES,fw_es            ;Initialize firmware ES register
            MOV   [DI].predef_val,3   ;Initialize the variables of the
            MOV   [DI].barrel_idx,6   ; config data area required by
            SHL   BL,1                ; config processing
            CMP   BL,2                ;Port ID is 2 for port 1
            JE    port_one
            MOV   BL,11               ;Port ID is 11 for port 2
port_one    LABEL NEAR
            MOV   [DI].port,BL
            MOV   AH,BL               ;Port identification
            MOV   AL,04H              ;04 for full duplex hardwired form
            PUSH  AX
            MOV   AX,0201H            ;01 for personality ID, 02 for config
                                      ; values are required
            PUSH  AX
            PUSH  DS                  ;Segment of scratch data area
            PUSH  DI                  ;Offset of scratch data area
            LEA   AX,[DI].port        ;Pointer to start of config data area
            PUSH  AX
            MOV   AX,3                ;Function code for CNP_GET_DEV_CONFIG
            PUSH  AX
            CALL  cnp_entry           ;Returns: AX = status, BX = fail reason
            CMP   AX,SUCCESS          ;Call error if failures
            MOV   AX,TRUE             ;Firmware call executed without errors
            JE    get_cf_ok
            MOV   AX,FALSE            ;Error during firmware call
get_cf_ok   LABEL NEAR
            RET
get_conf    ENDP
```

CHANGING THE DATA COMM CONFIGURATION

The configuration data structure initialized by the above function is used when reconfiguring the port with different baud rate, parity, handshaking and other parameters. The process of reconfiguration is as follows:

1. Save a copy of this data area for later restoration of the original configuration.

2. Change the appropriate structure parameters to reflect the desired new configuration.

3. Call *ver_conf* to "obtain the firmware device token", "get the firmware device", and verify that the new configuration data area is valid.

4. If the data is a valid configuration for the port (*ver_conf* returns TRUE), call *config* to "attach the firmware datacomm driver", reconfigure the hardware, "start the firmware driver", and "release the firmware datacomm device".

The following function performs the verification process, returning TRUE if the configuration data structure represents a valid port configuration.

```
;-------------------------------------------------------------------------
;  char ver_conf ( vc_scratch )
;
;     /* Verifies the datacomm configuration data in the config_data
;        structure.  Returns TRUE if configuration is valid, FALSE otherwise.
;        An example of an invalid configuration would be 8 data bits with
;        a parity other then "none".  The configuration should be checked
;        for validity prior to attempting to configure a port using the
;        config function.  The configuration data structure to be
;        verified is located at DS:'vc_scratch' */
;
vercf_args  STRUC                       ;Stack structure for passed arguments
            DW    ?                     ;Callers BP
            DW    ?                     ;Callers return address
vc_scratch  DW    ?                     ;'vc_scratch' argument
vercf_args  ENDS

            PUBLIC ver_conf
ver_conf    PROC  NEAR
            PUSH  BP
            MOV   BP,SP
            MOV   DI,[BP].vc_scratch ;'scratch' argument from stack
            POP   BP
            CALL  get_dtok           ;Obtain the firmware datacomm device token
            CALL  get_ddev           ;Get the firmware datacomm device
            MOV   ES,fw_es           ;Initialize firmware ES register
            PUSH  DS                 ;'vc_scratch' segment
            PUSH  DI                 ;'vc_scratch' offset
            MOV   AX,VERIFY_CODE     ;Firmware driver function code
            PUSH  AX
            PUSH  [DI].dev_token     ;Device token - set up using 'get_conf'
                                     ;  call prior to 'ver_conf'
            CALL  dhp_driver         ;Call firmware datacomm driver - returns
                                     ;  AX = status, BX = fail reason
            CMP   AX,SUCCESS
            JZ    conf_ok
            MOV   AX,FALSE           ;Configuration was bad
            JMP   vconf_done
conf_ok     LABEL NEAR
            MOV   AX,TRUE            ;Configuration was valid
vconf_done  LABEL NEAR
            RET
ver_conf    ENDP
```

```
get_dtok    PROC  NEAR                ;Obtain datacomm device token.  Saves
                                      ;  the token in the config data area.
            MOV   ES,fw_es            ;Initialize firmware ES register
            XOR   AX,AX
            MOV   AL,[DI].port
            PUSH  AX                  ;Port ID (2 = Port1, 11 = Port2)
            MOV   AX,DHP_GDTOK
            PUSH  AX                  ;Device handler function code
            CALL  dhp_entry           ;Token returned in BX
            MOV   [DI].dev_token,BX
            RET
get_dtok    ENDP

get_ddev    PROC  NEAR                ;Get the datacomm device allowing for
                                      ;  exclusive access.
            MOV   ES,fw_es            ;Initialize firmware ES register
            PUSH  [DI].dev_token
            MOV   AX,DHP_GDEV
            PUSH  AX
            CALL  dhp_entry
            RET
get_ddev    ENDP
```

The following function performs the actual port configuration and leaves the port operational from the firmware standpoint. It should be preceeded by the *ver_conf* call and only called if the *ver_conf* call returns TRUE.

```
;-------------------------------------------------------------------------
;  config ( cf_scratch )
;
;     /* Attaches the firmware datacomm driver, configures the datacomm port
;        using configuration data structure at DS:'cf_scratch', then starts
;        the firmware datacomm device driver for the port.
;        The attach call initializes the driver so that the configuration is
;        active.  It is necessary to start the datacomm device driver
;        following a configuration call in order to have the configuration
;        changes take effect.  The 'get_conf' function should be used to
;        initialize the configuration data area prior to modification of
;        variables in this area for the new configuration.
;        Returns TRUE if no error was encountered during calls to the
;        firmware, FALSE otherwise */
;
config_args STRUC                     ;Stack structure for passed arguments
            DW    ?                   ;Callers BP
            DW    ?                   ;Callers return address
cf_scratch  DW    ?                   ;'cf_scratch' argument
config_args ENDS
```

```
            PUBLIC config
config      PROC  NEAR
            PUSH  BP
            MOV   BP,SP
            MOV   DI,[BP].cf_scratch ;'cf_scratch' argument from stack
            POP   BP

                                     ;Attach the firmware datacomm driver -
                                     ;  initializes the driver so that the
                                     ;  configuration is active.
            MOV   ES,fw_es           ;Initialize firmware ES register
            XOR   AX,AX
            MOV   AL,[DI].port       ;Port id word (2 = Port1, 11 = Port2)
            PUSH  AX
            MOV   AX,2               ;Point-to-point firmware driver
            PUSH  AX
            MOV   AX,10              ;Firmware driver function code
            PUSH  AX                 ;  (DHP_ATTACH_DRIVER)
            CALL  dhp_entry
            CMP   AX,SUCCESS
            JNE   conf_error

                                     ;Configure the datacomm port.
            MOV   ES,fw_es           ;Initialize firmware ES register
            PUSH  DS                 ;'cf_scratch' segment
            PUSH  DI                 ;'cf_scratch' offset
            MOV   AX,CONFIG_CODE     ;Firmware driver function code
            PUSH  AX
            PUSH  [DI].dev_token
            CALL  dhp_driver         ;Always returns success

                                     ;Start the firmware datacomm driver
            MOV   ES,fw_es           ;Initialize firmware ES register
            MOV   AX,START_DRIVER_CODE
            PUSH  AX                 ;Firmware driver function code
            PUSH  [DI].dev_token     ;Device token from scratch area
            CALL  dhp_driver         ;Always returns success
            CALL  rel_ddev
            MOV   AX,TRUE            ;No error during attachment
            JMP   config_done

conf_error  LABEL NEAR
            MOV   AX,FALSE           ;Error while trying to attach driver
config_done LABEL NEAR
            RET
config      ENDP
```

```
rel_ddev    PROC  NEAR                ;Release the datacomm device
            MOV   ES,fw_es            ;Initialize firmware ES register
            PUSH  [DI].dev_token
            MOV   AX,DHP_RELEASE_DEVICE
            PUSH  AX                  ;Function code for firmware dev handler
            CALL  dhp_entry
            RET
rel_ddev    ENDP

PROG        ENDS

            END
```

RESTORING THE ORIGINAL CONFIGURATION

It is important for applications which modify a port's configuration to restore the original configuration prior to termination. Note that the above calls do not change the configuration reflected in the port configuration menus. If the ports are not restored to their originally configured state, their actual state will not correspond with that specified in the menus until the next system hard reset.

This applies to programs which access the data communication controller and associated hardware directly. Programs which modify configuration should include the following operations to ensure the hardware is reconfigured prior to program termination.

At program initialization:

1. Call *get_conf* to read the initial configured state.

2. If the program is going to use the *config* function to reconfigure the port, rather than going to the hardware directly, the configuration data area should be saved.

At program cleanup:

1. If the program used the *config* function to change the initial configuration, the configuration data area as saved at initialization should be restored.

2. The *ver_conf* function is called to obtain the firmware device token and get the firmware device. It also verifies the config data area as being valid (which it should be - it has not been changed).

3. The *config* function is called to actually restore the initial configuration as reflected by the config data area.

Data Comm Control Functions

The MS-DOS I/O Control For Devices System Function Call implements some special data comm control functions on the HP 150. Data communications data I/O using the MS-DOS Read and Write functions on COM1 or COM2 devices has been described earlier in this section. In addition to some special control functions, alternative and possibly preferable data I/O methods are described here.

IOCTL READS FOR INPUT

In addition to using the MS-DOS Read From a File/Device System Function (3FH) input may be accomplished through the I/O Control For Devices Function, request 2: "Read bytes from device control channel". Data channel bytes are returned for the COM devices. There are two principal advantages to input in this manner. First, the call gets any characters already received by the system without waiting for additional data (even if the requested number of bytes are not returned). Second, the call provides for optimal performance without having to put the device in "RAW" mode. The caller should check the AX register when MS-DOS returns (as specified for the function) to determine the number of bytes transferred.

An example of IOCTRL data input is shown below. It uses the *doscall* function previously described in this section.

```
#define IOC_READ 0x4402

int com_recv ( handle, buffer, length )

   /* Reads information from the datacomm device (COM1/COM2 previously
      opened, file handle 'handle') into a character buffer at 'buffer'.
      Will read a maximum of 'length' bytes.  Returns number of bytes
      actually read (0 for none read) */

int  handle;
char *buffer;
int length;

{
   doscall ( IOC_READ, handle, length, buffer );
   return ( (dc_ret.ah << 8) + dc_ret.al );
}
```

SPECIAL COM FUNCTIONS IMPLEMENTED THROUGH IOCTL WRITE REQUEST

The MS-DOS I/O Control for Devices System Function with Request 3: "Write to device control function" is used to implement a set of sub-functions for datacomm control. The defined sub-functions are:

(8,1) - Set 7-bit (normal) mode for both transmit and receive
(8,2) - Set 8-bit (binary) mode for both transmit and receive
(8,3) - Set transparent (monitor) mode
(8,4) - Disable transparent (monitor) mode
(8,5) - Modem disconnect
(8,6) - Send break
(8,7) - Fast buffer send
(8,8) - Reset data comm

The functions are specified with a two-byte identifier whose first byte is 8 to differentiate them from control requests to other devices. They are initiated by performing the MS-DOS I/O Control for Devices System Function with request 3. The buffer specified by the MS-DOS function (DS:DX) should contain the function code. For example, to send a break:

```
doscall ( IOC_WRITE, handle, length, &sub function );
```

where:

IOC WRITE is defined as 4403 Hex

and:

handle = device handle from device open call
length = 2 (sub-function number length)
sub function = 0807 Hex

Note that function (8,7) is a special case, described below.

The following definitions are in order:

BINARY MODE The eighth bit is not stripped from the data stream of received characters. In non-binary mode, either 7 or 8 bits are received as configured for the port (in the configuration menu). That is, when the port is configured for 7 data bit operation (in the config menu) , the eighth data bit (parity) is masked. In binary mode, the 7/8 data bits configured state is effectively overridden and a full 8 data bits are returned. The eighth bit will be the parity bit if the port is configured for 7 data bits in the config menu.

TRANSPARENT MODE In transparent mode, control characters normally resulting in some action being performed by the firmware datacomm driver are sent through as data instead. For example, if ENQ/ACK or XON/XOFF hanshaking for the port is enabled in the configuration menu for the port, placing the port in transparent mode effectively disables the handshaking from taking place. The handshaking settings in the configuration menus are not altered however. Both receive and transmit handshaking are effected. In addition, transparent mode causes parity checking to be disabled.

SEND BREAK The transmit data line is driven active (on) for 210 milliseconds. This is commonly used to signal a special condition to the host.

DISCONNECT MODEM Commonly used to cause a modem attached to the port to go off-line. The Terminal Ready control line output is driven inactive for two seconds.

RESET DATA COMM This function disables transparent mode and sets 7 bit (normal) mode. It has no other effect.

FAST BUFFER SEND FOR OUTPUT

For improved performance over the "Write to a File/Device" MS-DOS System Function for data output to the communications ports, a "fast send" I/O Control for Devices sub-function is offered. This sub-function is implemented in a manner similar to the "send break" call above however the buffer pointed to by DS:DX is slightly different. In addition to the sub-function number, three words follow it. The buffer structure for fast send is:

```
struct fcs_buf_tem {
                    int fcode;       /* sub-function number 0807 Hex */
                    int seq_offset;  /* send data buffer offset address */
                    int seq_segment; /* send data buffer segment address */
                    int count;       /* length of send data buffer (bytes) */
                    };
```

So a Microsoft C language compatible function to implement the fast send sub-function can be defined as follows:

```
com_send ( handle, buffer, length )

   /* Using "fast send buffer" function of IOCTRL for datacomm device to
      send 'length' bytes from 'buffer' to datacomm output channel with a
      handle of 'handle'. */

int handle;
char *buffer;
int length;

{
fcs_buf.fcode = 0x0807;
fcs_buf.seq_offset = buffer;
fcs_buf.seq_segment = dseg;
fcs_buf.count = length;

doscall ( IOC_WRITE, handle, 8, &fcs_buf );
}
```

where:

IOC_WRITE is defined as 4403 Hex

and:

handle = device handle from device open call
buffer = address (offset) of buffer containing send data
length = number of bytes in *buffer*

dseg is equal to the applications data segment register (DS)

HPIB INTERFACING

WARNING

This facility is meant for use with NON-DISC devices. Templates can interfere with the disc driver used by the MSDOS file system. In general, problems can occur if any action on the bus changes conditions at a disc drive. For example, since a UNIVERSAL DEVICE CLEAR resets the head on an 82905 drive, it should not be used. Let the user beware.

Limited HPIB Driver Functionality

It should be understood that the HP 150 does not implement the full set of HPIB functions through its firmware and as such has limited capabilities in an instrument-control environment. In general the firmware-based HPIB driver does not support and activity which may be initiated by a peripheral.

The computer peripherals on the HP 150 (discs, plotters, printers etcetera) are slave devices to the HP 150, which acts as a controller at all times. The HP 150 initiates all HPIB operations. The limitation is a firmware restriction, the HP 150 has a full hardware implementation of the HPIB standard. Therefore it may be necessary to write RAM-based drivers which deal directly with the HPIB controller hardware to support certain HPIB configurations.

Specifically the following conditions are not supported by the HP 150 firmware interface:

- SRQ (Service Request) Function
- TRIGGER Function
- Multiple controllers
- Transactions between other devices on the same bus

Opening the "HPIBDEV" Device

MSDOS functions are called by placing parameters in registers and executing a software interrupt 21 (hexadecimal). Controlling the HPIB device requires familiarity with two MSDOS functions. The "OPEN" function provides access to the HPIB driver. Once the driver is available, the "IOCTRL" function specifies the actions to be performed on the HPIB.

The OPEN function returns a "file handle" that is used by the IOCTRL function. To OPEN the HPIB device, set up the registers as follows and perform a software interrupt 21 hexadecimal.

```
AH      =  OPEN function number          = 61 (3Dh)
AL      =  access requested              = 0,1, or 2
DS,DX   -> the ASCII name of the device  = HPIBDEV
```

HPIB Control Calls

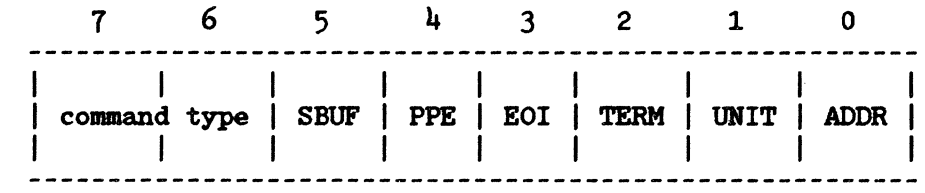
MS-DOS IOCTRL FUNCTION CALL. Now that the HPIB device is opened, the IOCTRL function is used to control the HPIB. The function IOCTRL requires:

```
AH      = IOCTRL function number          = 68 (44h)
AL      = subfunction number              = 3
BX      = file handle returned by OPEN function
CX      = count of bytes in the CONTROL BLOCK
DX,DX -> the CONTROL BLOCK
```

CONTROL BLOCK FORMAT. The CONTROL BLOCK has the following format:

Parm/Ret	SIZE	DESCRIPTION
Parm	word	Function high byte = 0 (numbers are in decimal) low byte = use of Control Block 17 = Perform this new template 18 = Continue this template --used to recover from a --parallel poll
Return	word	Status of function 0 = Success 4 = Busy (try NEW function again) 6 = Timeout Error 7 = EOI Error 8 = Bus Error 9 = Poll Error 10 = Undefined Error - HPIB bus protocol error. 11 = Overall Timeout Error
Parm	word	Data Buffer Length
Return	word	Number of data bytes sent or received
Parm	word	Offset of Data Buffer
Parm	word	Segment of Data Buffer
Parm	word	Offset of Template
Parm	word	Segment of Template
Return	word	Return Template Pointer
Parm	word	Overall Time Limit in milliseconds
Parm	byte	HPIB Device address
Parm	byte	HPIB Unit address

CONTROL TEMPLATE FORMAT. The template is a series of commands, and optionally, data bytes to the HPIB driver. Four types of commands exist with the high two bits of the command determining the type. The low 6 bits are qualifiers that further identify the command.



The following equates are used in the sample templates to denote command type.

Template Commands (bits 6,7)

SET_VALUE = 0 changes a configuration value in the HPIB driver.

RECEIVE = 040h indicates data should be read. The number of bytes read and where they are placed is determined by the lower 6 bits of the command. See the qualifiers described below. If the SBUF bit is clear, then one data byte is read and it is placed at the next location in the template. Regardless of the state of SBUF, the next byte is not considered a template command.

SEND = 080h indicates data should be sent. The number of bytes transmitted and where they are coming from is determined by the qualifiers.

CONTROLLER = 0C0H indicates the following byte (the controller command) should be transmitted with the attention (ATN) line high.

Some of the controller commands are:

UNTALK = 5Fh
UNLISTEN = 3Fh
TALK = 40h
LISTEN = 20H

The value of a command is created by "OR"ing the equates of the command types mentioned above with the following qualifiers.

NOTE

Terminator and time unit are carried from one template to another.

Qualifiers to Commands

----ADDRESS = 1 (bit 0)

When the CONTROLLER command has the address bit set, the drive address is "OR"ed with the following byte (the controller command). The address bit can be used with a SET VALUE command to specify what HPIB device address should be used for all the operations specified in the template, otherwise the address specified in the CONTROL BLOCK is used.

---- UNIT = 2h (bit 1)

When the CONTROLLER command has the unit bit set, the unit address is "OR"ed with the following byte (the controller command). The unit bit can be used with a SET VALUE command to change the HPIB unit address used with all the following template commands. (The default unit address is the one specified in the CONTROL BLOCK.)

---- TERM = 4h (bit 2)

The TERM bit implies the terminator character is of relevance when receiving characters. When used with SET VALUE, the following byte is the terminator character. When used with RECEIVE, the terminator character previously specified will denote the last character to be read.

NOTE

SET_VALUE + EOI + TRM => Read until TRM character, or EOI is received.

---- EOI = 8h (bit 3)

The EOI bit is used with RECEIVE to specify that with the last data byte read, an EOI is expected. If EOI expectancy state is not set correctly, an error is reported. Set Value + EOI => next byte in template is ignored and EOI should be ignored for the rest of the template.

---- PPE = 10h (bit 4)

PPE is used with SET VALUE to specify what devices the parallel poll should be looking for a response from. The following byte indicates the devices. PPE is used with disc drives. The PPE qualifier with RECEIVE, SEND, or CONTROLLER indicates a parallel poll will be performed before the command is executed for the current device id.

---- SBUF = 20h (bit 5)

The SBUF bit is used with RECEIVE or send to specify that a buffer rather than a single byte is to be transferred. The buffer offset address and the buffer segment address are always taken from the CONTROL BLOCK. When SET VALUE and SBUF are used, the next byte is the amount of time the HPIB driver waits for a single character before going into interrupt mode. The time unit is ~ 10 microseconds.

SAMPLE IDENTIFY TEMPLATES. On an identify command, two data bytes are expected with the EOI line asserted after the second data byte is received. The identify command is IDENTIFY_SECONDARY = 60h. The first template reads the data bytes into the buffer specified in the CONTROL BLOCK. The second template places the data bytes in the template itself with labels D1 and D2 allowing easy access to the data.

```
IDENTIFY TEMPLATE #1

     DB  CONTROLLER                    -  C0
     DB  UNLISTEN                      -  3F
     DB  CONTROLLER                    -  C0
     DB  UNTALK                        -  5F
     DB  CONTROLLER (or) ADDRESS       -  C1
     DB  IDENTIFY_SECONDARY            -  60
     DB  CONTROLLER                    -  C0
     DB  LISTEN + POC                  -  3E
     DB  RECEIVE (or) EOI (or) SBUF    -  68
     DB  0              - dummy byte   -  00
     DB  CONTROLLER                    -  C0
     DB  UNTALK                        -  5F
     DB  CONTROLLER                    -  C0
     DB  UNLISTEN                      -  3F
     DB  END_DATA     -end of template-  00

IDENTIFY TEMPLATE #2

     DB  CONTROLLER                    -  C0
     DB  UNLISTEN                      -  3F
     DB  CONTROLLER                    -  C0
     DB  UNTALK                        -  5F
     DB  CONTROLLER (or) ADDRESS       -  C1
     DB  IDENTIFY_SECONDARY            -  60
     DB  CONTROLLER                    -  C0
     DB  LISTEN + POC                  -  3E
     DB  RECEIVE                       -  40
D1:  DB  ?            -received byte   -   ?
     DB  RECEIVE (or) EOI              -  68
D2:  DB  ?            -received byte   -   ?
     DB  CONTROLLER                    -  C0
     DB  UNTALK                        -  5F
     DB  CONTROLLER                    -  C0
     DB  UNLISTEN                      -  3F
     DB  END_DATA     -end of template-  00
```

SAMPLE READ/WRITE BUFFER TEMPLATES. The following templates assume the buffer specified in the CONTROL BLOCK contains/receives the data.

BLOCK TRANSFER "IN" TEMPLATE

```
DB  SET_VALUE (or) TERM (or) EOI -  0C
DB  ?          -last data expected- FF
DB  CONTROLLER                   -  C0
DB  UNTALK                       -  5F
DB  CONTROLLER                   -  C0
DB  UNLISTEN                     -  3F
DB  CONTROLLER (or) ADDRESS      -  C1
DB  TALK                         -  40
DB  CONTROLLER                   -  C0
DB  LISTEN + POC                 -  3E
DB  RECEIVE (or) SBUF            -  60
DB  x                -dummy byte-   x
DB  CONTROLLER                   -  C0
DB  UNTALK                       -  5F
DB  CONTROLLER                   -  C0
DB  UNLISTEN                     -  3F
DB  END-DATA     -end of template-  00
```

BLOCK TRANSFER "OUT" TEMPLATE

```
DB  CONTROLLER                   -  C0
DB  UNTALK                       -  5F
DB  CONTROLLER                   -  C0
DB  UNLISTEN                     -  3F
DB  CONTROLLER (or) ADDRESS      -  C1
DB  LISTEN                       -  20
DB  CONTROLLER                   -  C0
DB  TALK + POC                   -  5E
DB  SEND (or) SBUF               -  A0
DB  x                -dummy byte-   x
DB  CONTROLLER                   -  C0
DB  UNTALK                       -  5F
DB  CONTROLLER                   -  C0
DB  UNLISTEN                     -  3F
DB  END_DATA     -end of template-  00
```

HPIB Interface Example (9111A Graphics Tablet)

The following is an example of using the MS-DOS I/O Control System Function accessing the "HPIBDEV" device to interface an HPIB peripheral. In this case it is a 9111A Graphics Tablet which is interfaced. The example is written in the C language.

The routines obtain and print stylus position and status information. The routines use the *doscall* function previously described (see "MS-DOS System Function Calls from the C Language"). The routine which returns the DS (data segment) register contents is described following the C code.

Note that this example for the sake of compactness contains minimal error checking. Error checking enhancements should be included for reliable functionality.

```
/*****************************************************************************

#define IO_DEVICE_CTRL          0x4403  /* MS-DOS I/O Control for Devices
                                           System Function code, sub-function
                                           3 (Write to Device Ctrl Channel ) */
#define CTRL_FUNCTION           17      /* Control Block function - perform
                                           this new template */
#define TIME_LIMIT              0x30    /* HPIBDEV IOCTRL Call time limit */
#define DEV_AND_UNIT_ADDRESS    0x0006  /* 9111A HPIB address code */

#define SET_VALUE               0x00    /* CONTROL TEMPLATE COMMANDS */
#define RECEIVE                 0x40
#define SEND                    0x80
#define CONTROLLER              0xc0

#define UNTALK                  0x5f    /* HPIB CONTROLLER COMMANDS */
#define UNLISTEN                0x3f
#define TALK                    0x40
#define LISTEN                  0x20

#define ADDRESS                 0x01    /* HPIB CONTROLLER COMMAND QUALIFIERS */
#define UNIT                    0x02
#define TERM                    0x04
#define EOI                     0x08
#define PPE                     0x10
#define SBUF                    0x20

#define END_DATA                0x00
#define POC                     0x1e
#define MAXLENGTH               128
#define LINEFEED                0x0a
#define EOF                     0xff
#define ESC 0x1B

int  tab_handle;
char buf[12];
```

```
struct  dc_ret_tem  {           /* 'doscall' return parameter structure */
                    char al;
                    char ah;
                    char bl;
                    char bh;
                    char cl;
                    char ch;
                    char dl;
                    char dh;
                    };

extern struct dc_ret_tem dc_ret;

static int hpib_ctrl_block[11] = {      /* Control block used for MS-DOS */
        CTRL_FUNCTION,                  /* I/O Control System Function call */
        0, 0,                           /* to "HPIBDEV" */
        0, 0,
        0, 0,
        0, 0,
        TIME_LIMIT,
        DEV_AND_UNIT_ADDRESS
        };

#define CTRL_BLOCK_COUNT        22      /* length of above block */

static char write_template[15] = {      /* Template for ASCII writes (the */
        CONTROLLER,                     /* only kind) to the tablet */
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        CONTROLLER + ADDRESS,
        LISTEN,
        CONTROLLER,
        TALK + POC,
        SEND + SBUF,
        0,
        CONTROLLER,
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        END_DATA
        };
```

```
static char read_binary[15] = {      /* Template for binary reads from tablet */

        CONTROLLER,
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        CONTROLLER + ADDRESS,
        TALK,
        CONTROLLER,
        LISTEN + POC,
        RECEIVE + SBUF + EOI,
        0,
        CONTROLLER,
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        END_DATA
        };

static char read_ascii[19] = {       /* Template for ASCII reads from tablet */
        SET_VALUE + EOI + TERM,
        LINEFEED,
        SET_VALUE + SBUF,
        EOF,
        CONTROLLER,
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        CONTROLLER + ADDRESS,
        TALK,
        CONTROLLER,
        LISTEN + POC,
        RECEIVE + SBUF + TERM,
        0,
        CONTROLLER,
        UNTALK,
        CONTROLLER,
        UNLISTEN,
        END_DATA
        };
```

```
/*****************************************************************************/
main ()
{
   int x, y, status;

   if ( tab_init() == 1 ) {              /* initialize graphics tablet */
      while ( kbhit() == 0 ) {           /* Continuously request stylus positi
                                            and status from tablet until
                                            keyboard is hit */
         get_x_y_status (&x, &y, &status, tab_handle);      /* get data */
         printf ("x: %x  y: %x  status: %x\r", x, y, status; /* display it */
      };
  };
xenixclose (tab_handle);                 /* close the tablet */
}

/*****************************************************************************/
char tab_init()

   /* Initializes the graphics tablet by opening the device and checking
      for the tablets response to the Identify function.  Returns 0 if the
      tablet did not identify correctly, returns 1 otherwise */

{
   tab_handle = open_tablet ();          /* open the tablet and get handle */
   write_block (tab_handle, "OI;", 3);   /* Perform tablet Output Identity */
   a_read (tab_handle, buf);             /* operation /*
   if (buf[0] != '9')                    /* Did tablet identify itself ? */
      printf ( "9111A Graphics Tablet did not identify\r\n" );
                                         /* Note - HPIB appears to "hang" when
                                            OI sent to a non-existant tablet */
      return ( 0 );
   }
   else
      return ( 1 );
}

/*****************************************************************************/
int open_tablet ()     /* Opens the device "HPIBDEV" and returns handle
                          Note - no error checking ! */
{
 hpib_ctrl_block[5] = hpib_ctrl_block[7] = data_seg ();
 return (xenixopen ("HPIBDEV", 2));
}
```

```
/*****************************************************************************/
int xenixopen (file_name, access)  /* open device "file_name" for mode
                                      "access " (read/write/both) */
char *file_name;
int access;

{
 int ax, dummy;

 ax = 0x3d00 + access;
 doscall (ax, dummy, dummy, file_name);
 return ( ( dc_ret.ah << 8 ) + dc_ret.al );  /* return handle */
}

/*****************************************************************************/
write_block (handle, buffer, count)   /* Writes "count" ASCII bytes from
                                         "buffer to device "handle" using
                                         MS-DOS Write to a File/Device
                                         System Function Call /*
char *buffer;

{
 hpib_ctrl_block[2] = count;                /* Update the control block  */
 hpib_ctrl_block[4] = (int) buffer;         /* with the passed parameters */
 hpib_ctrl_block[6] = (int) write_template;

 /* Perform the Write to a File/Device System Function Call */

 doscall (IO_DEVICE_CTRL, handle, CTRL_BLOCK_COUNT, hpib_ctrl_block);
}

/*****************************************************************************/
a_read (handle, buffer)  /* Reads a block of ASCII data from the specified
                            device. */
char *buffer;

{
 hpib_ctrl_block[2] = MAXLENGTH;           /* Update the control block with */
 hpib_ctrl_block[4] = (int) buffer;        /* passed parameters */
 hpib_ctrl_block[6] = (int) read_ascii;

/* Perform the read */

doscall (IO_DEVICE_CTRL, handle, CTRL_BLOCK_COUNT, hpib_ctrl_block);
}
```

```
/*************************************************************************/
get_x_y_status (x, y, status, tab_handle)

/* Reads the stylus position and status from the tablet.  Reads until six
   bytes of data are received.  Parses the address and status variables
   from the return string.  Note that due to timing considerations, only
   five bytes are received in some instances, followed by seven next time */

int *x, *y, *status;
int tab_handle;

{
 int my_x, my_y, my_status;

 while (b_read (&my_x, tab_handle) !=6);      /* Read until valid data */
 *x = ((my_x >> 8) & 0xff) + (my_x << 8);     /* Convert to 1byte, hbyte */
 *y = ((my_y >> 8) & 0xff) + (my_y << 8);
 *status = ((my_status >> 8) & 0xff) + (my_status << 8);
}

/*************************************************************************/
int b_read (buffer, handle)  /* Performs a binary data read from the tablet
                                until an EOI character is detected.  Returns
                                the number of bytes read from the tablet */
int *buffer;
int handle;

{
 hpib_ctrl_block[2] = 6;                 /* Update the control block with */
 hpib_ctrl_block[4] = (int) buffer;      /* passed parameters */
 hpib_ctrl_block[6] = (int) read_binary;

/* Perform the read */

doscall (IO_DEVICE_CTRL, handle, CTRL_BLOCK_COUNT, hpib_ctrl_block);
return (hpib_ctrl_block[3];
}

/*************************************************************************/
xenixclose (handle)   /* Close device with handle of "handle" */

int handle;

{
 doscall (0x3e00, handle);
}
```

```
;-------------------------------------------------------------------------
;  unsigned int data_seg()
;
   /* returns current DS register contents */

            PUBLIC data_seg
data_seg    PROC   NEAR
            MOV    AX,DS
            RET
data_seg    ENDP

PROG        ENDS
```

ACCESSORY CARD INTERFACING

Memory (Slot) Address Identification

16K of contiguous memory space is allocated to each of the two accessory slots on the HP 150. Associated with each slot is a hardware accessory card select line which goes active when a memory read or write occurs to the memory space allocated to that particular slot. The memory blocks are allocated to each slot as follows:

Accessory Slot 1	90000H - 93FFFH
Accessory Slot 2	A0000H - A3FFFH

Through utilization of the slot select line, accessory boards using memory mapped I/O space may be designed to be slot independent. That is, their memory mapped space will reside within the absolute bounds as defined for the particular slot in which the card is installed. A firmware routine called the Option Handling Processor (OHP) has a function which will return the slot ID (1 or 2) for a particular accessory, if that accessory is installed. Accessories with associated software wishing to utilize this function must have a unique id byte memory mapped to the base address of the 16K block. See the Accessories Subsystem discussion in Section 3 for more information.

The following example shows how to obtain the memory segment address for an installed card given its specific id.

```
ACC_ID      EQU   000H        ;Put your accessory board id number here
GET_TOKEN   EQU   007H        ;OHP get slot token function code
SLOT1       EQU   001H        ;Slot one id token
SLOT2       EQU   002H        ;Slot two id token
S1_SEG      EQU   09000H      ;Slot one RAM segment address
S2_SEG      EQU   0A000H      ;Slot two RAM segment address

;MSDOS entry interrupt and function codes

MSDOS       EQU   021H        ;MSDOS entry interrupt
OPEN_DEV    EQU   03DH        ;Open device function code
IOCNTRL     EQU   044H        ;I/O control for devices function code

;MS-DOS System Function parameter equates

INPUT       EQU   000H        ;Open device for input
OUTPUT      EQU   001H        ;Open device for output
READ        EQU   002H        ;Read from device
```

```
DGROUP      GROUP  DATA
DATA        SEGMENT WORD PUBLIC 'DATA'
            ASSUME DS:DGROUP

OHP_ENTRY   LABEL  DWORD          ;Option handler processor entry point
OHP_OSET    DW     000C0H         ;   Offset of option handler processor
OHP_SEG     DW     00040H         ;   Segment of option handler processor

HPIBDEV     DB     'HPIBDEV',0    ;Firmware device name string
FW_ES       DW     ?              ;Firmware ES value

DATA        ENDS
            PAGE
PGROUP      GROUP  PROG
PROG        SEGMENT BYTE PUBLIC 'PROG'
            ASSUME CS:PGROUP,DS:DGROUP,ES:NOTHING

;-------------------------------------------------------------------------
;    GET_ACC_SEG - Get accessory slot segment address for given id.  The
;                  firmware is called to locate which option slot contains
;                  the option board and the accessory slot RAM address
;                  segment pointer is set accordingly.
;
;         Registers in:  None
;         Registers out: AX = 0 if accessory board with given id not found
;                            Address base pointer (segment) if found
;         Registers preserved:  None

            PUBLIC GET_ACC_SEG
GET_ACC_SEG PROC   NEAR                    ;Initialize hardware interface
            CALL   GET_ES                  ;Load firmware ES value
            JNZ    INIT_ERR                ;Jump if error
            MOV    AX,ACC_ID               ;Load accessory board id number
            PUSH   AX
            MOV    AX,GET_TOKEN            ;Load get token function code
            PUSH   AX
            CALL   FAR PTR OHP             ;Call firmware option handler
            OR     AX,AX                   ;Jump if error
            JNZ    INIT_ERR
            MOV    AX,S1_SEG
            CMP    BX,SLOT1                ;Store option board RAM segment
            JNZ    CHK_S2                  ;   address depending on slot id
            RET
CHK_S2      LABEL  NEAR
            MOV    AX,S2_SEG
            CMP    BX,SLOT2
            JNZ    INIT_ERR
            RET

INIT_ERR    LABEL  NEAR                    ;Error occurred
            SUB    AX,AX                   ;Flag error occurred
            RET

GET_ACC_SEG ENDP
```

```
;-------------------------------------------------------------------------
;    OHP - Far call to firmware option handler processor.
;          This routine loads ES with the address of the firmware extra seg
;          and then does a long jump to the firmware OHP routine.  When
;          the OHP does a far return, control is returned to the caller
;          of this routine.
;          NOTE - The call to this routine must be a far call (32 bit)
;
;          Arguments in:
;               All arguments to the OHP are pushed on the stack and are
;               the callers responsibility.
;          Return values:
;               Values returned are specified by the OHP and depend on the
;               OHP function executed.
;          Registers preserved: None

            PUBLIC OHP
OHP         PROC   FAR                          ;Far call to firmware OHP
            MOV    ES,FW_ES                     ;Load ES for firmware
            JMP    DWORD PTR DGROUP:OHP_ENTRY ;Intersegment indirect jmp to OHP
            RET                                 ;Dummy return

OHP         ENDP

;-------------------------------------------------------------------------
;    GET_ES - Load and save firmware extra segment address.
;          This routine opens a firmware device 'HPIBDEV', reads four bytes
;          from it which is a 32-bit pointer to a firmware information block,
;          loads the firmware ES value from this information block, and
;          stores the ES value for later firmware calls.
;
;          Registers in:
;               None.
;          Registers out:
;               FF -  Z: everthing O.K., firmware ES value stored
;                     NZ: error in function calls, ES value not stored
;          Registers preserved:
;               BP, ES

GET_VARS    STRUC                               ;Stack structure for temp variables
FW_PTR      DD     ?                            ;Pointer to firmware info block
GET_VARS    ENDS

            PUBLIC GET_ES
GET_ES      PROC   NEAR                         ;Get and save firmware ES value
            PUSH   ES                           ;Save callers ES
            PUSH   BP                           ;Save callers BP
            SUB    SP,4                         ;Allocate space for temp variables
            MOV    BP,SP                        ;Initialize base pointer
            MOV    AH,OPEN_DEV                  ;Open device
            LEA    DX,HPIBDEV                   ;   name 'HPIBDEV'
            MOV    AL,INPUT                     ;   for input
            INT    MSDOS
            JC     GET_ERR                      ;Jump if error on open
```

```
            MOV     BX,AX                   ;Get device handle for read
            MOV     AH,IOCNTRL              ;I/O control for device
            MOV     AL,READ                 ;Read
            MOV     CX,4                    ;   four bytes
            LEA     DX,[BP].FW_PTR          ;   into temporary storage
            INT     MSDOS                   ;   from device handle in BX
            JC      GET_ERR                 ;Jump if error on read
            CMP     AX,4                    ;Jump if four bytes not read
            JNZ     GET_ERR
            LES     BX,[BP].FW_PTR          ;Load pointer to fw info block
            MOV     AX,ES:[BX]              ;Load firmware ES value
            MOV     FW_ES,AX                ;Store ES value
            ADD     SP,4                    ;Release allocated stack space
            POP     BP                      ;Restore callers BP
            POP     ES                      ;Restore callers ES
            SUB     AX,AX                   ;Flag everything O.K.
            RET                             ;Return to caller

GET_ERR     LABEL   NEAR                    ;Error occurred
            ADD     SP,4                    ;Release allocated stack space
            POP     BP                      ;Restore callers BP
            POP     ES                      ;Restore callers ES
            OR      AX,0FFFFH               ;Set error flag
            RET                             ;Return to caller

GET_ES      ENDP

PROG        ENDS

            END
```

AGIOS FUNCTION CALL REFERENCE

SECTION 8

This section is a reference to the Alpha Graphics Input/Output System (AGIOS) set of function calls on the HP 150.

AGIOS is a facility that lets you use system routines to perform tasks on the HP 150 keyboard and display. They let you perform text and graphics mode operations on your display, let you define and use softkeys (function keys), and let you perform all touch screen operations.

This manual contains information elsewhere pertinent to *using* the AGIOS function set. Section 5, System Software, provides an introduction to the Alpha Graphics I/O System and includes a simple example of an AGIOS function call. Section 7, Programming the HP 150, describes a general AGIOS caller function for the C language. This function provides a relatively easy way for C programs (and programs in other languages with suitable modifications to the *agios* function) to invoke any AGIOS function. Section 7 also includes many examples of AGIOS calls from the C language.

CONTENTS

SYNTAX USED IN THE AGIOS FUNCTION CALLS

Each AGIOS function call is explained in detail on the succeeding pages. In order to clarify the information that you need to supply, a standard notation is used. Parentheses and positional notation is used as follows:

PARM	Indicates a single byte parameter.
(PARM1,PARM2)	Indicates two single byte parameters with parm1 in the high byte and parm2 in the low byte.
(,PARM)	Indicates a single byte parameter in the low order byte of the word. The high byte is ignored.
(PARM,)	Indicates a single byte parameter in the high order byte of the word. The low byte is ignored.
(PARM)	Indicates a word (16 bit) parameter.
((PARM))	Indicates a double word parameter. Usually the first word is a data segment address and the second word is an offset address.

Where applicable, the AGIOS call name is followed by the corresponding escape sequence. For example, one entry later in this section is:

DEFINE TOUCH FIELD (ESC - z g)

This indicates that the escape sequence which corresponds to the Define Touch Field AGIOS call is ESC - z g.

BATCH FUNCTION CALL

A special function call is available which lets you execute a sequence of function calls automatically. Using this function call is especially convenient when you frequently perform the same set of AGIOS function calls.

To "batch" function calls, you set up the sequence of AGIOS function calls in a ***command buffer***. Then you issue the following batch function call using the command buffer as one of its parameters.

(0, 0)	Function code
(BUFFER LENGTH)	COMMAND BUFFER length (byte count).
((COMMAND BUFFER))	A pointer to the buffer containing the AGIOS function calls. The function calls are defined consecutively. Use the same parameter format as specified in this section for the individual function calls.

A batch call is aborted when any of the function calls in the batch causes an error condition. Additionally, you cannot nest batch function calls (include them in the batch).

Example:

This example clears the alpha display by homeing up first, then clearing the display. Refer to the "H" and "J" options of the "Execute Two Character Sequence" function call.

The command buffer looks like this:

Contents-->	*16*	*0*	*H*	*16*	*0*	*J*
Byte -->	*+0*	*+1*	*+2*	*+3*	*+4*	*+5*

The assembler routine that sets up the command buffer and executes a batch call might look like this:

```
CLS     PUSH DS                 ; Save DS on Stack
        POP  CMDSEG             ; And Store it in Batch Buffer
;
        MOV  AX,4403H           ; I/O Control Write
        MOV  BX,1               ; Console Handle
        MOV  CX,8               ; Batch Buffer Length
        MOV  DX,offset BATCH    ; Batch Command Buffer
        INT  21H
        RET
;
CMDBUF DB   16,0,'H',16,0,'J'
;
BATCH   DB   0,0                ; AGIOS Batch Command
BUFLEN  DW   6                  ; CMDBUF Len 6 Here
CMDOFF  DW   CMDBUF             ; CMDOFF equates CMDBUF
CMDSEG  DW   0                  ; Data Segment Dummy Value
```

VIDEO INTRINSICS

The video intrinsics are a set of functions that may be used to update the state of the display. With the exception of the Write Line function, all intrinsics operate on a pre-defined subset area of the 24 by 80 character display. All row and column values are relative to zero. The upper left corner of the display is (0,0) and the lower right corner is (23,79).

A null data buffer pointer (segment = 0FFFFH) will suppress the update operation for that data type. There is a one to one correspondence between the position of a data byte in its buffer and the character position that it will affect in the pre-defined update area, starting at the upper left corner, incrementing column position first and then row position.

The ASCII data consists of the 8 bit HP Standard ASCII character code. The character set data consists of character set code characters as follows:

CHARACTER CODE:	*CHARACTER SET SELECTED:*
@	*Normal Roman*
A	*Line Drawing*
B	*Bold Face Roman*
C	*Italic Roman*
D	*Math*
SPACE	*No Change*

The enhancement data consists of enhancement code characters as follows:

security off:	*@*	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*K*	*L*	*M*	*N*	*O*
security on :	*P*	*Q*	*R*	*S*	*T*	*U*	*V*	*W*	*X*	*Y*	*Z*	*[*	**	*]*		
half-bright									*x*	*x*	*x*	*x*	*x*	*x*	*x*	*x*
underline					*x*	*x*	*x*	*x*					*x*	*x*	*x*	*x*
inverse video			*x*	*x*			*x*	*x*			*x*	*x*			*x*	*x*
blinking		*x*		*x*		*x*		*x*		*x*		*x*		*x*		*x*

(A space, 20H, indicates no change to the current state.)

DEFINE AREA

This function specifies the area to be operated upon by subsequent area update operations.

(0, 1)	Function Code.
(LR-ROW,LR-COL)	Defines the lower right corner of the area to be operated upon.
(UL-ROW,UL-COL)	Defines the upper left corner of the area to be operated upon.
((PREV-COORD))	A pointer to a buffer where the previous area coordinates are returned. The format of the returned data is the same as the two input coordinates. This buffer pointer may be null.

WRITE AREA

This function writes data into the pre-defined display area.

(0, 2)	Function Code
(DATA LENGTH)	Length of the data buffers.
((ENH POINTER))	A double word pointer to a buffer of enhancement code characters to be used to change the enhancement state of the update area.
((CHAR SET))	A double word pointer to a buffer of character set code characters to be used to change the character set state of the update area.
((ASCII POINTER))	A double word pointer to the buffer of ASCII data to be written into the update area.

CLEAR AREA

This function clears the most recently defined area to ASCII blanks (20H), with no enhancements set.

(0, 3)	Function Code

ENHANCE AREA

This function sets the enhancement state of the entire pre- defined display area to the specified enhancement.

(0, 4)	Function Code
(,ENHANCEMENT)	An enhancement code character.

READ AREA

This function reads data from the pre-defined display area.

(0, 5)	Function Code
(DATA LENGTH)	Length of the data buffers.
((ENH POINTER))	A double word pointer to the buffer that the enhancement data will be read into.
((CHAR SET))	A double word pointer to a buffer that the character set data will be read into.
((ASCII POINTER))	A double word pointer to the buffer that the ASCII data will be read into.

SHIFT AREA

This function shifts the data in the pre-defined display area. Enhancements and character set are shifted with the ASCII data. Data shifted off an edge of the update area is lost.

(0, 6)	Function Code
(DATA LENGTH)	Length of the data buffer.
((ENH POINTER))	A double word pointer to a buffer of enhancement codes to be used to enhance the remaining unshifted area.
((CHAR SET))	A double word pointer to a buffer of character set code characters to be used to change the character set state of the remaining unshifted area.
((ASCII POINTER))	A double word pointer to a buffer of ASCII data to be written into the remaining unshifted area.
(DIRECTION,DIST)	DIRECTION: 0 = up 1 = down 2 = left 3 = right DIST: The number of rows/columns to shift the current video data.

WRITE LINE

This function writes a single row (or part of a row) in the workspace. Unlike Write Area, this intrinsic ignores the area bounds set by Define Area. If the position and length of the data are defined such that the right workspace boundary is violated, that portion of the data exceeding the boundary is ignored. No line wrap occurs.

(0, 7)	Function Code
(WKSP-ROW,WKSP-COL)	Defines the workspace relative position at which the data will be written.
(DATA LENGTH)	Length of the data buffers.
((ENH POINTER))	A double word pointer to the buffer of enhancement data to be written at the designated position.
((CHAR SET))	A double word pointer to a buffer of character set code characters to be used to change the character set state of the update area.
((ASCII POINTER))	A double word pointer to the buffer of ASCII data to be written at the designated position.

APPLICATION SOFTKEYS

This section gives the AGIOS function calls that support Application Softkeys. You can access these softkeys by typing [Shift] [User System]. You can use ASCII and Extended Roman characters within Application Softkey labels.

UPDATE SOFTKEY LABEL

This function will update a softkey label and enhancement.

(0, 8)	Function Code
(,NUMBER)	Softkey Number (the softkey number is from 1 to 8 inclusive)
((DATA))	A double word pointer to the buffer of ASCII data to be written into the label area. (16 bytes.)
(,TOP ENH)	Enhancement code for the top half of the label.
(,BOT ENH)	Enhancement code for the bottom half of the label.

READ SOFTKEY LABEL

This function gets the softkey number specified by the caller and then returns the softkey label and the enhancement code characters in two buffers.

(0, 9)	Function Code
(,NUMBER)	Softkey Number
((DATA))	A double word pointer to the buffer which is for the ASCII data.
((ENHANCEMENTS))	A double word pointer to the buffer which is for the enhancement code characters.

DISPLAY SOFTKEY LABELS

This function displays the application softkey labels in the softkey window.

(0,11)	Function Code

CONTROL FUNCTIONS

EXECUTE TWO CHARACTER SEQUENCE (ESC CHAR)

(0,16) Function Code

OP-CHAR The character equivalent of a 2 character escape sequence. Any operation characters are valid except for those that return data. The following list defines some of the most common ones.

0	:	*Dump Alpha to Printer*	*U*	:	*Next Page*
1	:	*Set tab*	*V*	:	*Previous Page*
2	:	*Clear tab*	*W*	:	*Format Mode On*
3	:	*Clear all tabs*	*X*	:	*Format Mode Off*
4	:	*Set left margin*	*Y*	:	*Display Functions On*
5	:	*Set right margin*	*Z*	:	*Display Functions Off*
9	:	*Clear margins*	*[*	:	*Start Unprotected Field*
@	:	*Delay one second*	*]*	:	*End Protected Field*
A	:	*Cursor up*	*b*	:	*Enable Keyboard*
B	:	*Cursor down*	*c*	:	*Disable Keyboard*
C	:	*Cursor right*	*f*	:	*Modem Disconnect*
D	:	*Cursor left*	*g*	:	*Soft Reset Terminal*
E	:	*Reset terminal*	*h*	:	*Home Up*
F	:	*Home down*	*i*	:	*Back Tab*
G	:	*Return*	*j*	:	*Display Softkey Def Menu*
H	:	*Home up*	*k*	:	*Exit Softkey Def Menu*
I	:	*Tab*	*l*	:	*Memory Lock On*
J	:	*Clear display*	*m*	:	*Memory Lock Off*
K	:	*Clear line*	*p*	:	*Default [f1] Value*
L	:	*Insert Line*	*q*	:	*Default [f2] Value*
L	:	*Delete Line*	*r*	:	*Default [f3] Value*
P	:	*Delete Character w/o Wrap*	*s*	:	*Default [f4] Value*
Q	:	*Insert Character w/o Wrap*	*t*	:	*Default [f5] Value*
R	:	*Insert Character Off*	*u*	:	*Default [f6] Value*
S	:	*Roll Up*	*v*	:	*Default [f7] Value*
T	:	*Roll Down*	*w*	:	*Default [f8] Value*

POSITION CURSOR (ESC & a)

(0,17)	Function Code
MODE	*Bit 0:* *1 = window row address* *0 = workspace row address* *Bit 1:* *1 = relative row address* *0 = absolute row address* *Bit 2:* *1 = negative row address* *0 = positive row address* *Bit 3:* *1 = row address is valid* *0 = row address is not valid* *Bit 4:* *1 = window column address* *0 = workspace column address* *Bit 5:* *1 = relative column address* *0 = absolute column address* *Bit 6:* *1 = negative column address* *0 = positive column address* *Bit 7:* *1 = column address is valid* *0 = column address is not valid*
(COLUMN)	An unsigned integer
(ROW)	An unsigned integer

DEFINE ENHANCEMENTS (ESC & d)

(0,18)	Function Code
SEC	SEC: 1 = on, 0 = off.
ENH	ENH: the ESC &d code character (@..O)

CURSOR SENSE ABSOLUTE (ESC a)

(0,19)	Function Code
((BUFFER))	A pointer to a buffer where two words are returned. The first word is the column number in binary and the second word is the row number in binary.

CURSOR SENSE RELATIVE (ESC `)

(0,20)	Function Code
((BUFFER))	A pointer to a buffer where two words are returned. The first word is the column number number in binary and the second word is the row number in binary.

SET CURSOR TYPE

This function sets the alpha cursor type.

(0,21)	Function Code
(,TYPE)	Alpha cursor type: 0 = underscore 1 = inverse cell.

READ CURSOR TYPE

This function reads the alpha cursor type.

(0,22)	Function Code
((BUFFER))	A pointer to a word location where alpha cursor type data is stored.

READ TERMINAL CONFIGURATION

This function reads the current terminal configurations.

(0,24)	Function Code
((BUFFER))	A word pointer to the buffer where the current configuration is returned.

When this function is complete, BUFFER contains:

(,RRRRRTSP)	R = reserved bits, T = set if touch screen off, S = set if softkeys on, P = set if remote port 2.
(KEYBOARD LANGUAGE)	
(STRING LANGUAGE)	
(OP SYS DEVICE)	Bits 0-2: addr. 0-7. Bits 3-15: dev. 0 = HP-IB, 1 = Accsy.

TOUCH SCREEN FUNCTIONS

The touch features on the HP 150 can be programmed in a variety of ways. The two general types of touch operations are "Field" operations and "Row/Column" operations. These two types can be intermixed.

Several of the touch screen function calls in this section assume keycode mode. Refer to "Keycode Modes" in Section 7 for information on this mode.

FIELD OPERATIONS

There are four types of touch fields you can define. They are:

ASCII Fields:

This mode is very similar to the User-Definable Softkeys (see Section 4). A buffer of ASCII characters is associated with a touch field. A response string of 0 to 80 ASCII characters is obtained by consecutive keyboard input operations. The first input obtains the first ASCII byte, and the second input obtains the second ASCII byte, etc. The response string is generated when the field is touched and should be indistinguishable from the typing of the same string from the keyboard. Auto-repeat is performed.

Keycode Fields:

Keycode fields require that you be in keycode mode (see "Keycodes", Section 7). The two data words of the response string are treated as a keycode and a qualfier and are processed by the regular keyboard routines. The final response to touch depends on the state and mode of keyboard processing. Touch simulates typing on the keyboard. Releasing simulates releasing your finger from the key. Auto-repeat is performed.

Toggle Fields:

The touch field is defined as a toggle switch. Touching the area toggles the field on and off. Whenever the field is touched, sensing information is passed to the application. The sensing information consists of three data bytes. The data is obtained by three consecutive keyboard input operations. The qualifier word of each data byte returned to the application has the touch screen ID. The three data bytes of sensing information are:

01H - toggle on field report opcode
d1 - response string first byte
d2 - response string second byte

02H - toggle off field report opcode
d1 - response string first byte
d2 - response string second byte

Normal Fields:

This type of touch field senses touch and/or release. The sensing information consists of three data bytes. The data can be obtained by three consecutive keyboard input operations. The qualifier word of each data byte returned to the application has the touch screen ID. Auto-repeat is performed. The three data bytes of sensing information are:

05H - field touched report opcode
d1 - response string first byte
d2 - response string second byte

06H - field released report opcode
d1 - response string first byte
d2 - response string second byte

Touch fields can overlap. If they do, then the most recent definition for a character cell takes precedent.

ROW COLUMN OPERATIONS

This type of touch operation returns the row and column position when a touch occurs. The row and column position are returned byte-by-byte using the keyboard input function of the operating system. Three data bytes are returned. The qualifier word returned with each byte of data has the touch screen ID. The data bytes for row/column operations are:

03H - row column touch report opcode
row - touched row number in binary
col - touched column number in binary

The data bytes for release report of row/column are:

04H - row column release report opcode
row - touched row number in binary
col - touched column number in binary

DEFINE TOUCH FIELD (ESC - z g)

(0,32)	Function code
((STRING))	Pointer to response string. Points to 2 words for keycode field the first word is the qualifier and the second word is the keycode. Points to 2 bytes for toggle or normal field, 0 - 80 bytes for ASCII field.
(LENGTH)	Response string length
(ATTRIBUTE,MODE)	Touch ATTRIBUTE: 1= ASCII field 2= Keycode field 3= Toggle field 4= Normal field Reporting MODE: 1= Report when touched 2= Report when released 3= Report both touch and release
(ON-ENH,OFF-ENH)	Enhancements of the field for on and off state for toggle field. Also enhancements of the field when touched and released for normal, ASCII, and keycode fields.
(CURSOR,BEEP)	CURSOR: 0 = do not position cursor 1 = position cursor on touch BEEP: 0 = do not beep 1 = beep on touch
(LR-ROW,LR-COL)	Row and column of the lower right corner of the touch field.
(UL-ROW,UL-COL)	Row and column of the upper left corner of the touch field.

DEFINE SOFTKEY FIELD (ESC - z s)

This function defines one of the eight softkey label areas as a touch field. These fields when touched produce the same response as if the corresponding function key is typed. The default is all softkey touch fields are on.

(0,33)	Function code
(MODE,KEY)	KEY (Softkey number): 1-8 MODE: 1 = on, 0 = off.

DELETE TOUCH FIELD (ESC - z d)

Deletes the touch field with upper left corner at <row> <col>. Nothing happens if there is no touch field there. The row and column are screen relative coordinates.

(0,34)	Function code
(UL-ROW,UL-COL)	Row and column position of the field to be deleted. (0FFH,0FFH) deletes all fields.

TOUCH SCREEN RESET (ESC - z j)

Resets all fields to off.

(0,35)	Function code

SET TOUCH REPORTING MODES (ESC - z n)

This function determines if, and how, touch is reported to your application by the HP 150 terminal.

(0,36) Function code

(,SCREEN-MODE) Touch Field and Row/Col sensing:

- 0 - Disable reporting.
- 1 - Enable sensing for row/column position. Touch fields are inactive.
- 2 - Enable sensing for touch fields only. Row/column sensing is inactive.
- 3 - Enable sensing for both row/column and touch fields. Row/column sensing occurs for areas not defined as touch fields.
- 4 - Toggles touch screen on and off.
- 10-14 - Same as 0-4, but causes escape sequence reports to be sent. This form is used ONLY by the system parser.

(,TOUCH-MODE) Sense touch or touch-release: (used with Row/Col sensing only)

- 1 - Report on Touch
- 2 - Report on Release
- 3 - Report on both Touch and Release

KEYBOARD INTERCEPT

Keyboard Intercept functions let you gain more control over the use of the keyboard. Each of the keyboard keys can be individually set to normal processing or one of the special processing modes. It should be noted that the keycodes for [f1] through [f12] are valid only when the application softkey labels are displayed on the screen with AGIOS function call (0,11).

There is no explicit "get keycode and qualifier" function call. The standard operating system console input function returns the normal ASCII code and also keycodes. The qualifiers are also returned if keycode mode is on.

The qualifier word is composed of the following bit values:

Bit	Value
15-8	Input Device ID: 0C0H = keyboard 080H = touch screen 000H = terminal internal
7	Special key. If set, the data is a non-ASCII keycode.
6	Reserved.
5	Left extend char - set when down.
4	Right extend char - set when down.
3	Control - set when down.
2	Left shift - set when down.
1	Right shift - set when down.
0	Repeating key when set.

To properly use the key intercept functions, the application program should first put the operating system's console input device into raw mode. This will allow keycodes to be passed through without interpretation by the operating system. Keycode mode should then be turned on. Finally each of the keys on the keyboard can be set to the desired mode of operation. For more information see "Keyboard Interfacing" in Section 7 of this manual.

DEFINE KEY CHARACTERISTICS

This function lets you alter characteristics of any of the special keys. Specifically, you can:

- Process the key normally (Same as on HP 2623 terminal)
- Intercept the key and pass a keycode to the application for processing
- Ignore the key when it is pressed
- Beep when the key is pressed in combination with the above characteristics

(0,40) Function Code

(CHARACTERISTICS) Key Characteristics

Bit:	Action: 0	beep
1	intercept	
2	ignore	
3-15	reserved	

If bit 1 and 2 are both set, the key is treated as an intercept key. When both are zero, the key resumes normal functioning.

(KEYCODE) Keycode from table on previous page. A value of 0FEH will set all special keys to the specified characteristics.

GET KEY CHARACTERISTICS

This function returns the characteristics of a key to the caller.

(0,41) Function Code

((BUFFER)) Pointer to a buffer where the key's characteristics will be returned.

(KEYCODE) Keycode

PUT KEY

This function lets you specify direct the terminal to process the keycode normally. Use this function when you wish to 'process' a keypress which was read in when intercept mode was active. For normal ASCII keys, you would simply 'echo' the character in place of using this function.

(0,42) Function Code

(QUALIFIER) Qualifier

Bit: Interpretation:

15-8 = Input Device ID (0C0H = keyboard)
7 = Special key. Must be set.
6 = Reserved
5 = Left extend char, set when down
4 = Right extend char, set when down
3 = Control, set when down
2 = Left shift, set when down
1 = Right shift, set when down
0 = Not used

(KEYCODE) Keycode

KEYCODE ON/OFF

This function turns the keycode mode of the console device on and off. If keycode mode is off, each key hit on the keyboard returns one byte of data when the console input device is read. If keycode mode is on, each key press returns four bytes of data. The first two bytes form a word of qualifiers and the next two bytes form a word of key data. See '*Keycode Modes*' in Section 7 for more detail.

(0,43) Function Code

MODE Keycode mode:

1 = on
0 = off

KEYCODE STATUS

This function returns the on/off status of a keycode.

(0,44)	Function Code
((BUFFER))	A pointer to a byte location where the keycode on/off status is returned.

READ KEYPAD STATUS

This function returns the status of whether the extended keypad is in numeric or graphics mode.

(0,44)	Function Code
((BUFFER))	A pointer to a byte location where the keypad status is returned.

When this function is complete, BUFFER contains 0 if numeric mode is set, and 1 if graphics mode is set.

DISPLAY CONTROL (ESC * d)

CLEAR GRAPHICS MEMORY (ESC * d a)

This function clears graphics display memory to 0. The complete displayable graphics area of 512 x 390 dots are cleared.

(4, 1) Function Code

SET GRAPHICS MEMORY (ESC * d b)

This function sets graphics display memory to 1. The complete displayable graphics area of 512 x 390 dots are set.

(4, 2) Function Code

TURN ON GRAPHICS DISPLAY (ESC * d c)

This function turns on the graphics display. The data in graphics memory is not affected.

(4, 3) Function Code

TURN OFF GRAPHICS DISPLAY (ESC * d d)

This function turns off the graphics display. The data in graphics memory is not affected.

(4, 4) Function Code

TURN ON ALPHANUMERIC DISPLAY (ESC * d e)

This function turns on the alphanumeric display and alphanumeric cursor. The data in alphanumeric memory is not affected.

(4, 5) Function Code

TURN OFF ALPHANUMERIC DISPLAY (ESC * d f)

This function turns off the alphanumeric display. The data in alphanumeric memory is not affected.

(4, 6) Function Code

TURN ON GRAPHICS CURSOR (ESC * d k)

This function turns on the graphics cursor. The data in graphics memory is not affected.

(4, 7) Function Code

TURN OFF GRAPHICS CURSOR (ESC * d l)

This function turns off the graphics cursor. The data in graphics memory is not affected.

(4, 8) Function Code

TURN ON RUBBER BAND LINE (ESC * d m)

This function turns on the rubber band line and graphics cursor.

(4, 9) Function Code

TURN OFF RUBBER BAND LINE (ESC * d n)

This function turns off the rubber band line.

(4,10) Function Code

MOVE GRAPHICS CURSOR ABSOLUTE (ESC * d <x,y> o)

This function moves the graphics cursor to the specified location. The move occurs even if the cursor is turned off.

(4,11) Function Code

(X-COORD) The X coordinate of the new cursor position expressed as an absolute number in the range of plus and minus 16383.

(Y-COORD) The Y coordinate of the new cursor position expressed as an absolute number in the range of plus and minus 16383.

MOVE GRAPHICS CURSOR INCREMENTAL (ESC * d <x,y> p)

This function moves the graphics cursor to the specified location. The move occurs even if the cursor is turned off.

(4,12) Function Code

(X-COORD) The X coordinates of the new cursor position expressed as a number that is relative to the current cursor position. Its range extends from -32768 to +32767.

(Y-COORD) The Y coordinate of the new cursor position expressed as a number that is relative to the current cursor position. Its range extends from -32768 to +32767.

TURN ON ALPHANUMERIC CURSOR (ESC * d q)

This function turns on the alphanumeric cursor. The data in alphanumeric memory is not affected.

(4,13) Function Code

TURN OFF ALPHANUMERIC CURSOR (ESC * d r)

This function turns off the alphanumeric cursor. The data in alphanumeric memory is not affected.

(4,14) Function Code

TURN ON GRAPHICS TEXT MODE (ESC * d s)

This function turns on graphics text mode. Characters that normally go to the alphanumeric display will be drawn on the graphics display.

(4,15) Function Code

TURN OFF GRAPHICS TEXT MODE (ESC * d t)

This function turns off graphics text mode.

(4,16) Function Code

VECTOR DRAWING MODE (ESC * m)

SELECT DRAWING MODE (ESC * m <mode> a)

This function selects the vector drawing mode.

(4,17) Function Code

(MODE) Drawing Mode:

0 = Graphics memory not changed
1 = Clear mode
2 = Set mode
3 = Complement mode
4 = Jam mode

SELECT LINE TYPE (ESC * m <type> b)

This function selects the vector line type.

(4,18) Function Code

(TYPE) Line Type:

1 = _____
2 = User defined line pattern
3 = Current area fill pattern
4 = — - — - — - — - — - — -
5 = ____ ____ ____ ____ ____
6 = — — — — — — — —
7 = ...
8 = —·—·—·—·—·—·—·—·—·
9 = --- --- --- --- --- --- --- --- --- ---
10= __ _ _ __ _ - - — - - — - —
11= Point plot

DEFINE LINE PATTERN AND SCALE (ESC * m <pattern><scale> c)

This function defines a user line pattern and scale.

(4,19)	Function Code
(,PATTERN)	The line pattern expressed as an eight bit binary number.
(SCALE)	The line scale expressed as a binary number from 1 to 16.

DEFINE AREA FILL PATTERN (ESC * m <pattern> d)

This function defines a user area fill pattern of 8 by 8 screen dots.

(4,20)	Function Code
(,DATA ROW1) (,DATA ROW2) (,DATA ROW3) (,DATA ROW2) (,DATA ROW4) (,DATA ROW5) (,DATA ROW6) (,DATA ROW7) (,DATA ROW8)	You specify all eight rows of the 8 by 8 fill pattern. Each DATA-ROW is an eight-bit byte which defines a particular row of the pattern.

FILL RECTANGULAR AREA, ABSOLUTE (ESC * m <x1,y1,x2,y2> e)

This function fills a rectangular area with the selected line or area fill pattern. The rectangular region is defined by specifying the lower left and upper right coordinates.

(4,21)	Function Code
(LWR LEFT X-COORD)	Each coordinate is in the range of -16384 to +16383.
(LWR LEFT Y-COORD)	
(UPR RIGHT X-COORD)	
(UPR RIGHT Y-COORD)	

FILL RECTANGULAR AREA, RELOCATABLE (ESC * m <x1,y1,x2,y2> f)

This function fills a rectangular area with the selected line or area fill pattern. The rectangular region is defined by specifying the lower left and upper right coordinates.

(4,22)	Function Code
(LWR LEFT X-COORD)	Each value is in the range from -32768 to +32767.
(LWR LEFT Y-COORD)	
(UPR RIGHT X-COORD)	
(UPR RIGHT Y-COORD)	

SELECT POLYGONAL FILL PATTERN (ESC * m <pattern> g)

This function selects a pattern for polygonal and rectangular area fill.

(4,23) Function Code

(PATTERN) Area Fill Pattern:

1 = Solid fill pattern
2 = User-defined fill pattern
3-10 = Pre-defined fill pattern

SELECT BOUNDARY PEN (ESC * m <pen> h)

This function selects the pen to be used to draw the boundary of a filled polygon. The actual value is not significant in a black and white system. This function turns on boundary drawing with a solid line pattern. The drawing of each edge of the boundary can be individually controlled.

(4,24) Function Code

(PEN) Boundary Pen Number

NO POLYGON BOUNDARY (ESC * m h)

This function turns off drawing of boundary around a polygon.

(4,25) Function Code

SET RELOCATABLE ORIGIN (ESC * m <x,y> j)

This function sets the relocatable origin to the specified absolute location.

(4,26)	Function Code
(X-COORD)	The X coordinate is the new relocatable origin expressed as an absolute number in the range of -16384 to +16383.
(Y-COORD)	The Y coordinate is the new relocatable origin expressed as an absolute number in the range of -16384 to +16383.

SET RELOCATABLE ORIGIN TO PEN POSITION (ESC * m k)

This function sets the relocatable origin to the current pen position.

(4,27)	Function Code

SET RELOCATABLE ORIGIN TO CURSOR POSITION (ESC * m l)

This function sets the relocatable origin to the current cursor position.

(4,28)	Function Code

GRAPHICS TEXT (ESC *)

The HP 150 offers a comprehensive graphics character set in read-only memory (*ROM*). This standard character set is used by all graphics text operations. However, you do have the ability to create custom characters of your own design and to use these singly or to replace the entire built-in character set.

SET GRAPHICS TEXT SIZE (ESC * m <size> m)

This function sets the graphics text size. The vector lists that define the current character set are scaled using this text size.

(4,29)	Function Code
(X-SCALE)	The X coordinate scale factor for text characters. The format is a 16 bit number with the radix point between bits 7 and 8: Bits 1-8 = integer Bits 9-16 = fraction
(Y-SCALE)	The Y coordinate scale factor for text characters. The format is a 16 bit number with the radix point between bits 7 and 8: Bits 1-8 = integer Bits 9-16 = fraction

SET GRAPHICS TEXT ORIENTATION (ESC * m <orientation> n)

This function selects the graphics text orientation. This also changes the direction of line feed, carriage return, and backspace. The desired orientation is specified by a number defined as:

(4,30) Function Code

(ORIENTATION) Graphics Text Orientation:

1 = Normal
2 = Rotate 90 degrees counterclockwise
3 = Rotate 180 degrees counterclockwise
4 = Rotate 270 degrees counterclockwise

TURN ON TEXT SLANT (ESC * m o)

This function turns on the 26.57 degree slant of graphics text characters.

(4,31) Function Code

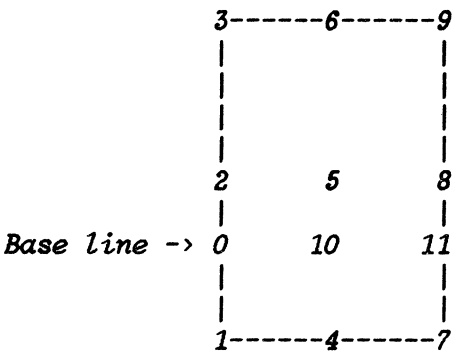
TURN OFF TEXT SLANT (ESC * m p)

This function turns off 26.57 degrees slant of graphics text characters.

(4,32) Function Code

SET GRAPHICS TEXT ORIGIN (ESC * m <0-9> q)

This function sets the graphics text origin to one of twelve positions of text justification. The positions are shown in this figure:



(4,33) Function Code

(ORIGIN) Graphics Text Origin:

A number from 0 to 11.

GRAPHICS TEXT LABEL (ESC * l <text>)

This function outputs a string of graphics characters.

(4,34) Function Code

((TEXT)) Segment and offset address of a string of characters. The string must be terminated by CR, LF, CR LF, or LF CR.

DEFINE USER CHARACTER SET

This function lets you re-define the entire graphics character set. All subsequent graphics text operations will use this character set. This includes text size, orientation, slant, and justification.

(4,35)	Function Code
((TABLE))	Segment and offset address of the table that points to the vector lists of characters.

SELECT DEFAULT CHARACTER SET

This function sets the character set to the default set maintained by the system. The cell size is 7 x 10.

(4,36)	Function Code

OUTPUT SINGLE TEXT CHARACTER

This function outputs a single graphics character defined by a vector list. All current graphics text operations such as size and orientation apply.

(4,37)	Function Code
((CHARACTER))	Segment and offset address of the vector list of a single character.

SET GRAPHICS DEFAULT (ESC * m r)

This function sets the graphics parameters to their default values.

(4,38) Function Code

The defaults affected by this call are:

- Pen down
- Line type 1
- User-defined line pattern solid
- User-defined area fill pattern solid
- Boundary pen off
- Drawing mode set
- Relocatable origin 0,0
- Text size 1
- Text origin 1
- Text slant off
- Text orientation 1
- Graphics text off
- Graphics display on
- Alphanumeric display on
- Graphics cursor off
- Alphanumeric cursor on
- Rubber band line off
- Graphics cursor position 0,0

SET PICTURE DEFINITION DEFAULTS (ESC * m 1 r)

This function sets the picture definition parameters to their default values.

(4,72)	Function Code
(RESET LEVEL)	The level of graphics reset. On the HP 150 the value '1' is the only supported level.

The picture defaults are:

Pen down
Line type 1
User-defined line pattern solid
User-defined area fill pattern solid
Boundary pen off
Drawing mode set
Text size 1
Text origin 1
Text slant off
Text orientation 1
Graphics text off

GRAPHICS HARD RESET (ESC * w r)

Sets the graphics parameters to their power on state.

(4,73)	Function Code

GRAPHICS PLOTTING (ESC * p)

LIFT PEN (ESC * p a)

This function lifts the pen.

(4,39)	Function Code

VECTOR MOVE (ESC * p a <x,y>)

This function lifts the pen and moves the pen to the new coordinate position. The pen is lowered at the end of the operation.

There are three ways to specify the new coordinate position:

(4,40)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give an absolute coordinate position in the range from -16384 to +16383.
(4,41)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give an incremental coordinate position in the range from -32768 to +32767.
(4,42)	Function Code
(X-coord) (Y-coord)	The X and Y numbers give a relocatable coordinate position in the range from -32768 to +32767.

LOWER PEN (ESC * p b)

This function lowers the pen.

(4,43)	Function Code

VECTOR DRAW (ESC * p b <x,y>)

This function lowers the pen and draws a vector to the new coordinate position. The pen is lowered at the end of the operation.

There are three ways to specify the new vector coordinates:

(4,44)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the absolute coordinates of the vector position. They are in the range from -16384 to +16383.
(4,45)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the incremental coordinates of the vector position. They are in the range from -32768 to +32767.
(4,46)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the relocatable coordinates of the vector position. They are in the range from -32768 to +32767.

PLOT TO CURSOR POSITION (ESC * p c)

This function moves the pen from its current position to the current cursor position if the pen is up. A draw is performed from the current pen position to the current cursor position if the pen is down.

(4,47) Function Code

POINT PLOT (ESC * p d)

This function draws a dot at the current pen position and then lifts the pen.

(4,48) Function Code

SET RELOCATABLE ORIGIN TO PEN POSITION (ESC * p e)

This function sets the relocatable origin to the current pen position.

(4,49) Function Code

START POLYGONAL AREA FILL (ESC * p s)

This function starts polygonal area fill. The boundary pen is lowered with this function.

(4,50) Function Code

TERMINATE POLYGONAL AREA FILL (ESC * p t)

This function terminates the polygon definition and fills the polygon.

(4,51) Function Code

POLYGON MOVE

This function closes the polygon defined up to this point and moves the pen to the new coordinate position to start a new polygon.

There are three ways to specify the new coordinate position:

(4,52) Function Code

(X-COORD)
(Y-COORD) The X and Y numbers give the absolute coordinates of the new position. They are in the range from -16384 to +16383.

(4,53) Function Code

(X-COORD)
(Y-COORD) The X and Y numbers give the incremental coordinates of the new position. They are in the range from -32767 to +32767.

(4,54) Function Code

(X-COORD) (Y-COORD)	The X and Y numbers give the relocatable coordinates of the new position. They are in the range from -32768 to +32767.

POLYGON DRAW

This function defines the edge of a polygon from the current pen position to the new coordinate position.

There are three ways that you can specify the new coordinate position:

(4,55)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the absolute coordinates of the new position. They are in the range from -16384 to +16383.
(4,56)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the incremental coordinates of the new position. They are in the range from -32768 to +32767.
(4,57)	Function Code
(X-COORD) (Y-COORD)	The X and Y numbers give the relocatable coordinates of the new position. They are the range from -32768 to +32767.

LIFT BOUNDARY PEN (ESC * p u)

This function lifts the polygon boundary pen. Undrawn edges of the polygon are not drawn. This remains in effect until the boundary pen is lowered.

(4,58)	Function Code

LOWER BOUNDARY PEN (ESC * p v)

This function lowers the polygon boundary pen. If a boundary pen has been specified, undrawn edges of the polygon are drawn with a solid line pattern. This remains in effect until the boundary pen is lifted.

(4,59) Function Code

GRAPHICS STATUS (ESC * s)

READ DEVICE ID (ESC * s 1)

This function returns the device id of the HP 150.

(4,60)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for returned device data.

When this function is complete, BUFFER contains an ASCII string that identifies the device.

READ PEN POSITION (ESC * s 2)

This function returns the current position and the state of the pen.

(4,61)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for returned pen status data.

When this function is complete, BUFFER contains:

(X-COORD) (Y-COORD)	The binary X and Y coordinates of the current pen position
(STATE)	0 = pen lifted, 1 = pen lowered

READ CURSOR POSITION (ESC * s 3)

This function returns the current position of the cursor.

(4,62)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned cursor data.

When this function is complete, BUFFER contains:

(X-COORD) (Y-COORD)	The X and Y coordinates of current cursor position.

READ CURSOR POSITION, WAIT FOR KEY (ESC * s 4)

This function returns the current position of the cursor, but lets the user move it on the display first. The user can type any ASCII key or the SELECT key on the keyboard to move the cursor. As soon as one of these characters is typed, the cursor coordinates are returned to the program.

(4,63)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for returned cursor position.

When this function is complete, BUFFER contains:

(X-COORD) (Y-COORD)	The X and Y coordinates of current cursor position.
(CODE)	The character code of the key that was typed.

READ DISPLAY SIZE (ESC * s 5)

This function returns the number of displayable units and also the number of units in millimeters.

(4,64)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned displayable size and unit data.

When this function is complete, BUFFER contains:

(X-LWR-LEFT) (Y-LWR-LEFT) (X-UPR-RIGHT) (Y-UPR-RIGHT)	The X and Y coordinates of the maximum display size.
(X-MM) (Y-MM)	The X and Y dimensions in dots / millimeters.

READ GRAPHICS SETTINGS (ESC * s 6)

This function returns information about the current graphics settings in effect.

(4,65)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for returned graphics settings.

When this function is complete, BUFFER contains the settings in consecutive words:

(CLEAR DISPLAY)
(NUMBER OF PENS)
(RESERVED)
(RESERVED)
(AREA SHADING)
(RESERVED)
(RESERVED)
(DYNAMIC MODIFICATION)
(GRAPHICS CHARACTER SIZE)
(GRAPHICS CHARACTER ANGLES)

(GRAPHICS CHARACTER SLANT)
(DOT-DASH LINE PATTERN)
(RESERVED)
(RESERVED)
(RESERVED)
(RESERVED)

READ GRAPHICS TEXT STATUS (ESC * s 7)

This function returns the current attributes of graphics text.

(4,66)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned graphics attributes.

When this function is complete, BUFFER contains:

(X SIZE) (Y SIZE)	The character cell size.
(ORIGIN)	The text origin.
(ANGLE)	The text orientation.
(SLANT)	The character slant.

READ ZOOM STATUS (ESC * s 8)

This function returns the terminal's zoom setting.

(4,67)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned zoom setting.

When this function is complete, BUFFER contains:

(ZOOM SIZE)	1 - 16 (the HP 150 always returns 1).
(ZOOM ON/OFF)	0 = off, 1 = on (the HP 150 always returns 0).

READ RELOCATABLE ORIGIN (ESC * s 9)

This function returns the current relocatable origin.

(4,68)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned origin.

When this function is complete, BUFFER contains:

(X-COORD) (Y-COORD)	The X and Y coordinates of the current relocatable origin.

READ RESET STATUS (ESC * s 10)

This function returns information on whether the terminal has executed a full reset since the last time reset status was checked.

(4,69)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned reset status.

When this function is complete, BUFFER contains:

(RESET STATUS)
(RESERVED)
(RESERVED)
(RESERVED)
(RESERVED)
(RESERVED)
(RESERVED)

(RESERVED)

READ AREA SHADING (ESC * s 11)

This function returns information on the area shading capability of the terminal.

(4,70)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned shading data.

When this function is complete, BUFFER contains:

(CAPABILITIES)	The area shading capabilities.
(WIDTH) (HEIGHT)	The area shading pattern size.

READ DYNAMICS (ESC * s 12)

This function returns information on the terminal's dynamic graphics capabilities.

(4,71)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned dynamic graphics data.

When this function is complete, BUFFER contains:

(SELECTIVE-ERASE-CAPABILITIES)

(COMPLEMENT-CAPABILITIES)

READ EXTENDED SCREEN DIMENSIONS

This function provides information about the alpha and graphics screen size plus the relationship between the two.

(0,74)	Function Code
((BUFFER))	Segment and offset address of the buffer to be used for the returned screen size data.

When this function is complete, BUFFER contains:

(X-PIXELS)	512 graphics display size in pixels.
(Y-PIXELS)	390
(ROWS)	27 alpha display size.
(COLUMNS)	80
(X-MM)	160 graphics display size in mm.
(Y-MM)	120
(ROW-MM)	150 alpha display size in mm.
(COL-MM)	116
(DELTA-X)	10 graphics origin minus alpha origin in mm.
(DELTA-Y)	4

LOGIC DIAGRAMS

APPENDIX

A

Schematic diagrams for the HP 150 are provided in this appendix.

Table of Contents

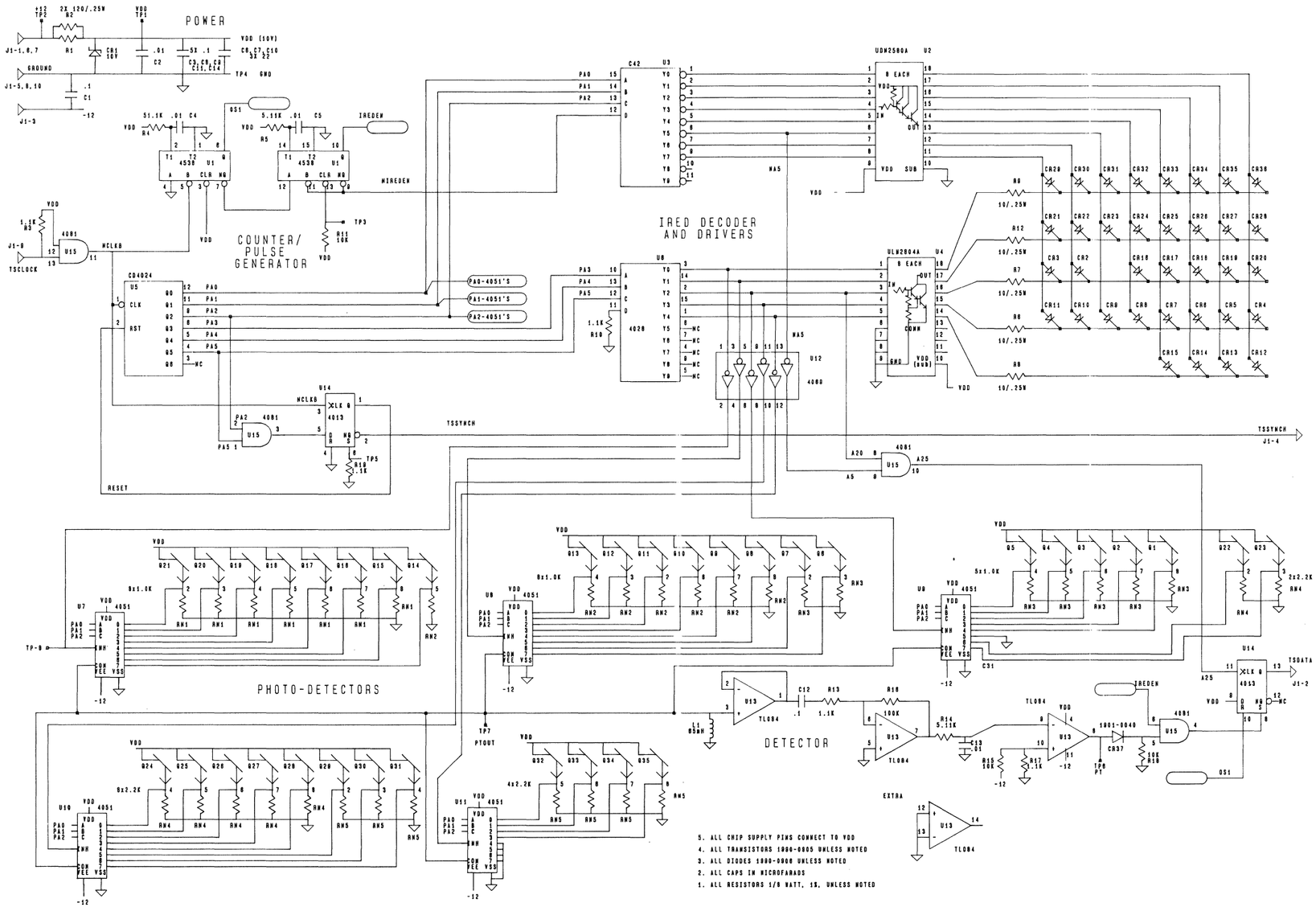
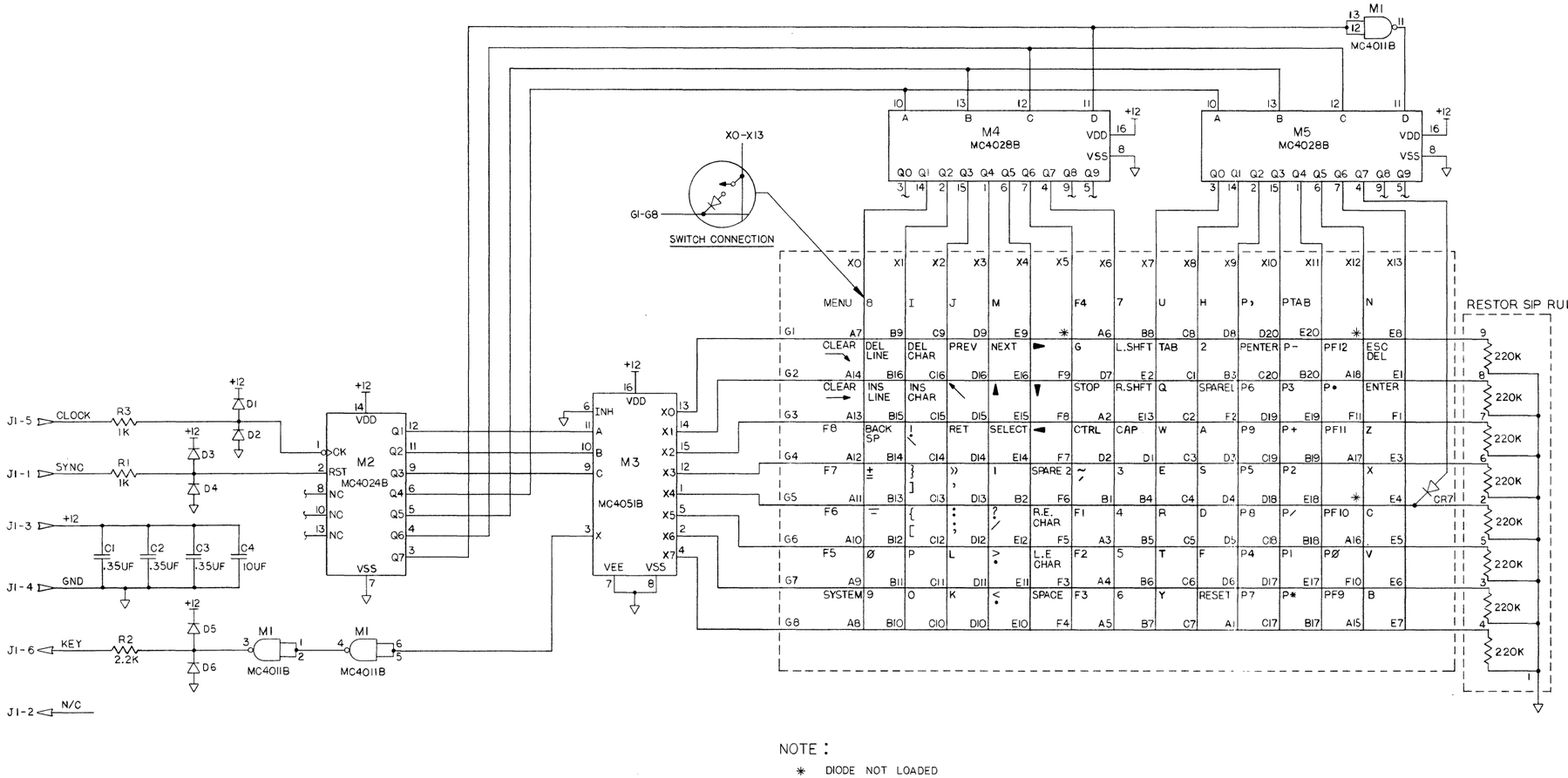


Figure A-1. Touchscreen PCA

This page is blank



NOTE :

* DIODE NOT LOADED

Figure A-2. Keyboard PCA

This page is blank

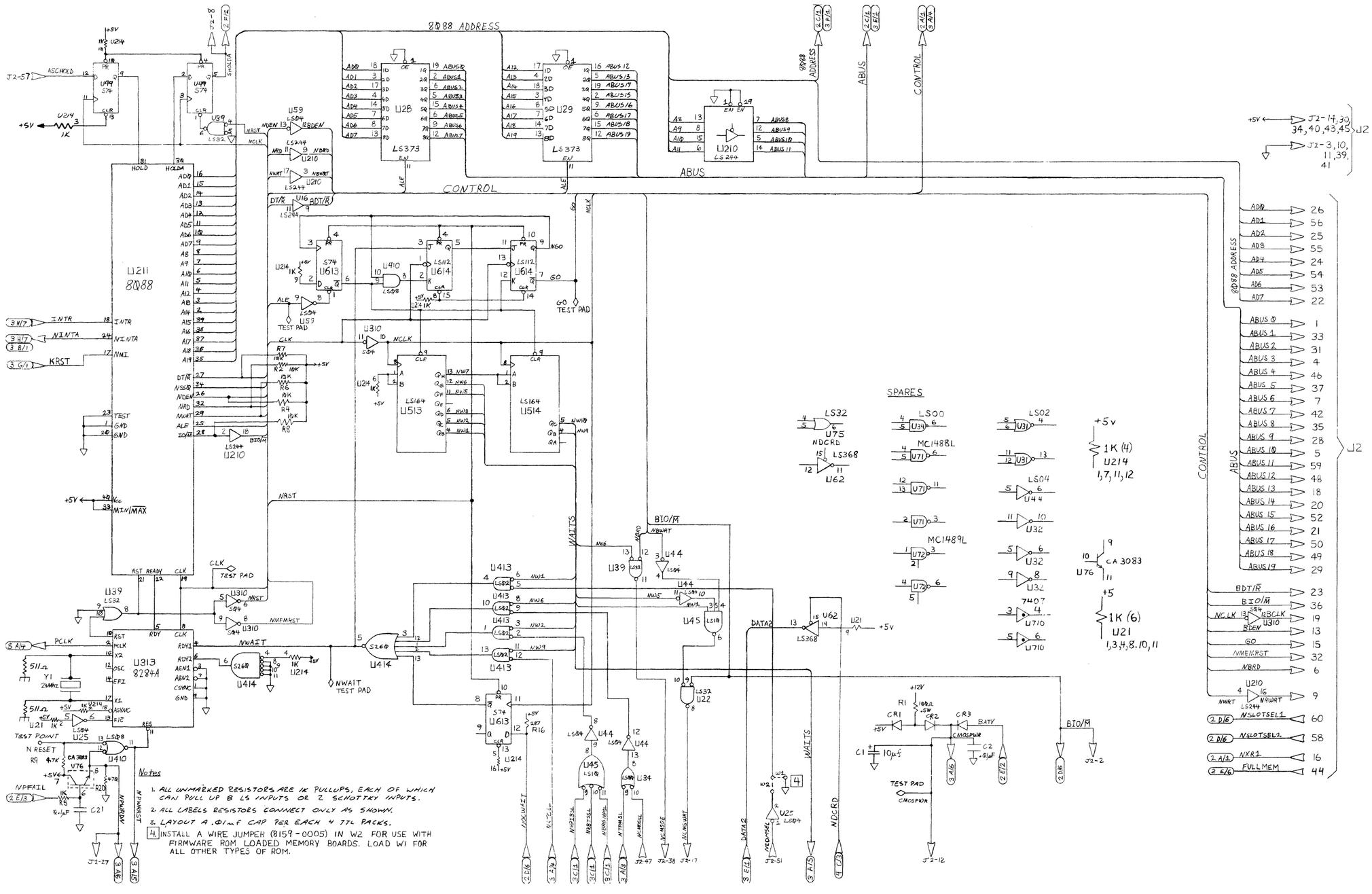


Figure A-3. Processor PCA

This page is blank

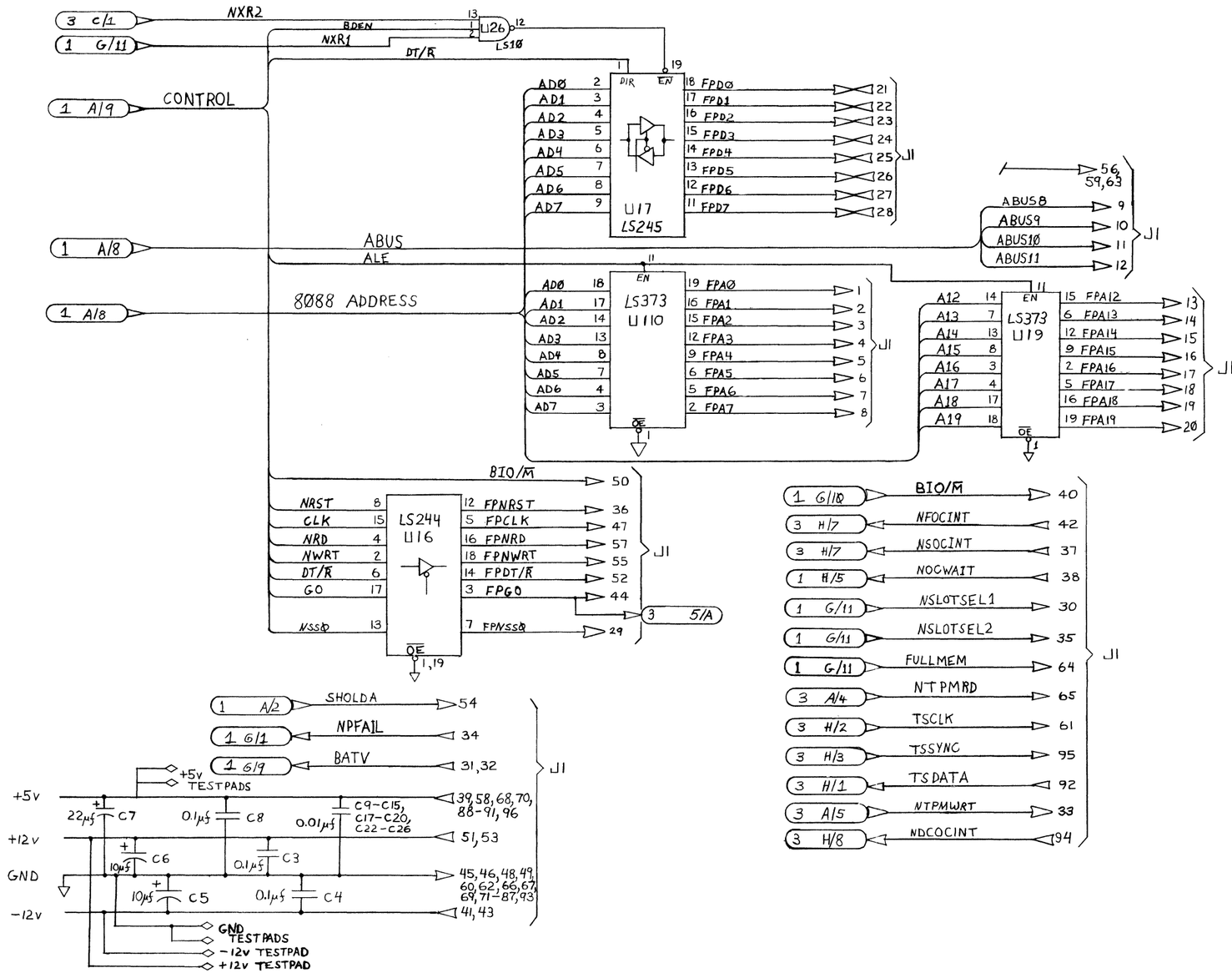


Figure A-4. Processor Front Plane PCA

This page is blank

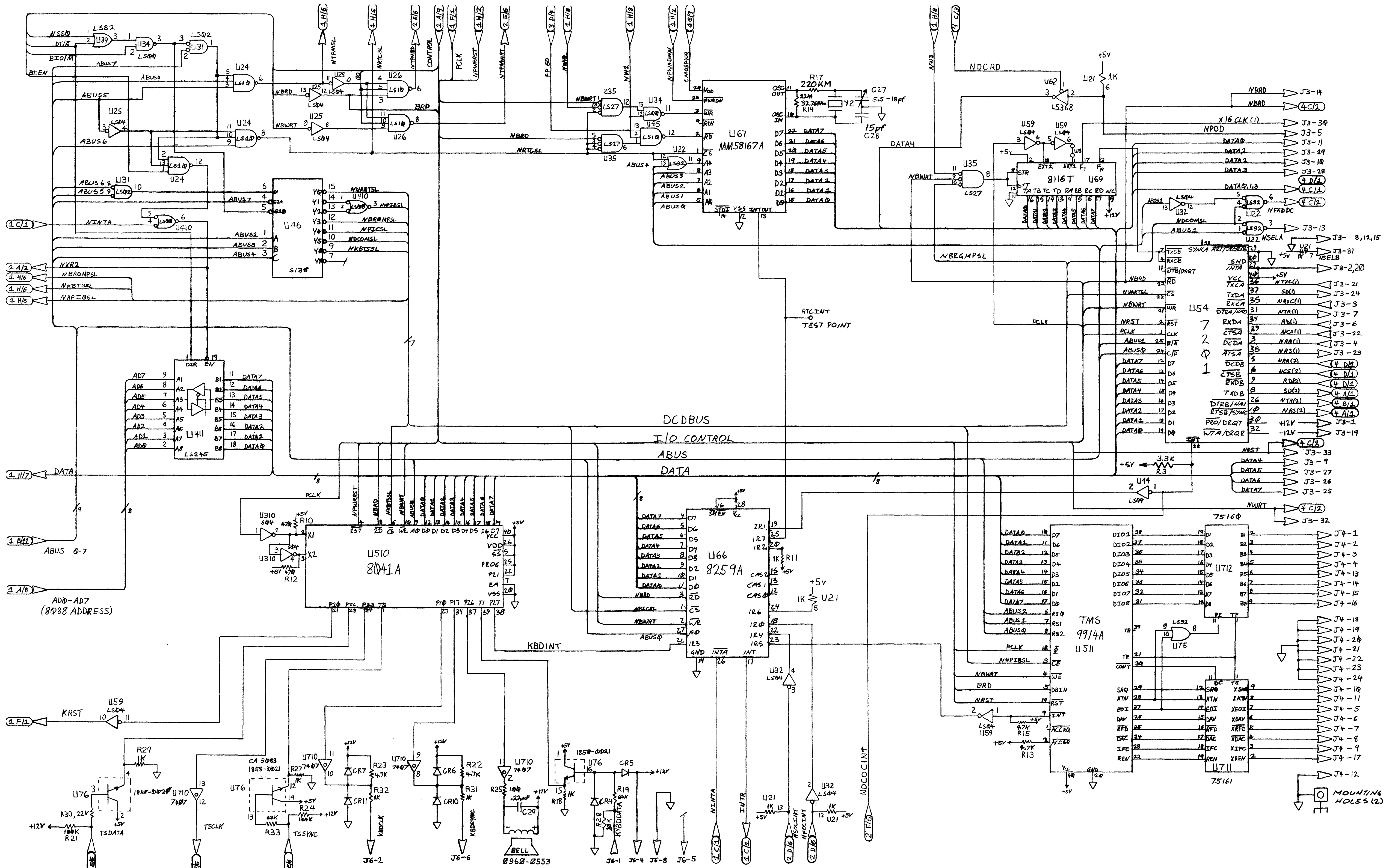
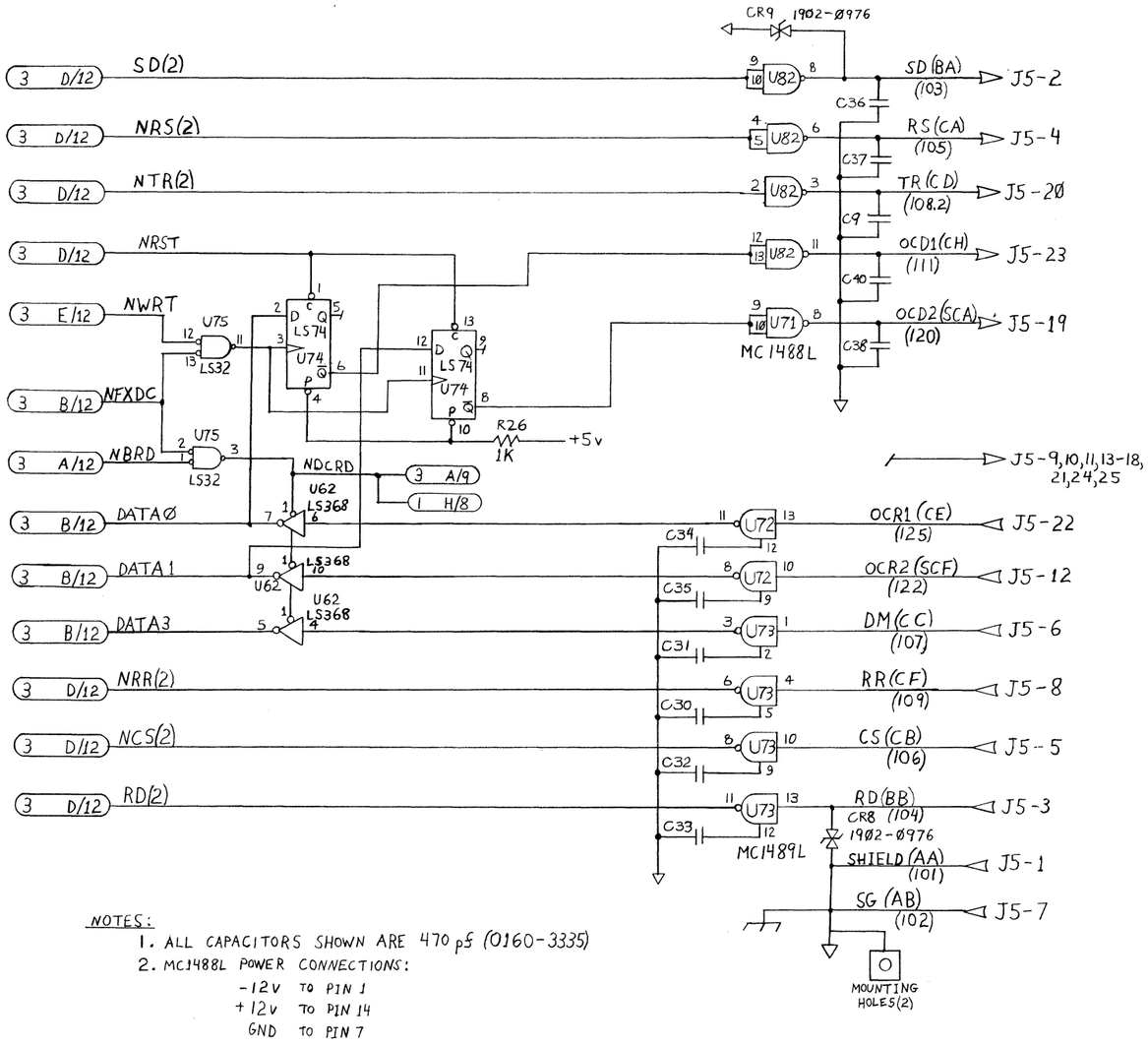


Figure A-5. Processor I/O Bus

This page is blank



NOTES:

1. ALL CAPACITORS SHOWN ARE 470 pf (0160-3335)
2. MC1488L POWER CONNECTIONS:
   - -12v TO PIN 1
   - +12v TO PIN 14
   - GND TO PIN 7

Figure A-6. Keyboard PCA Port

This page is blank

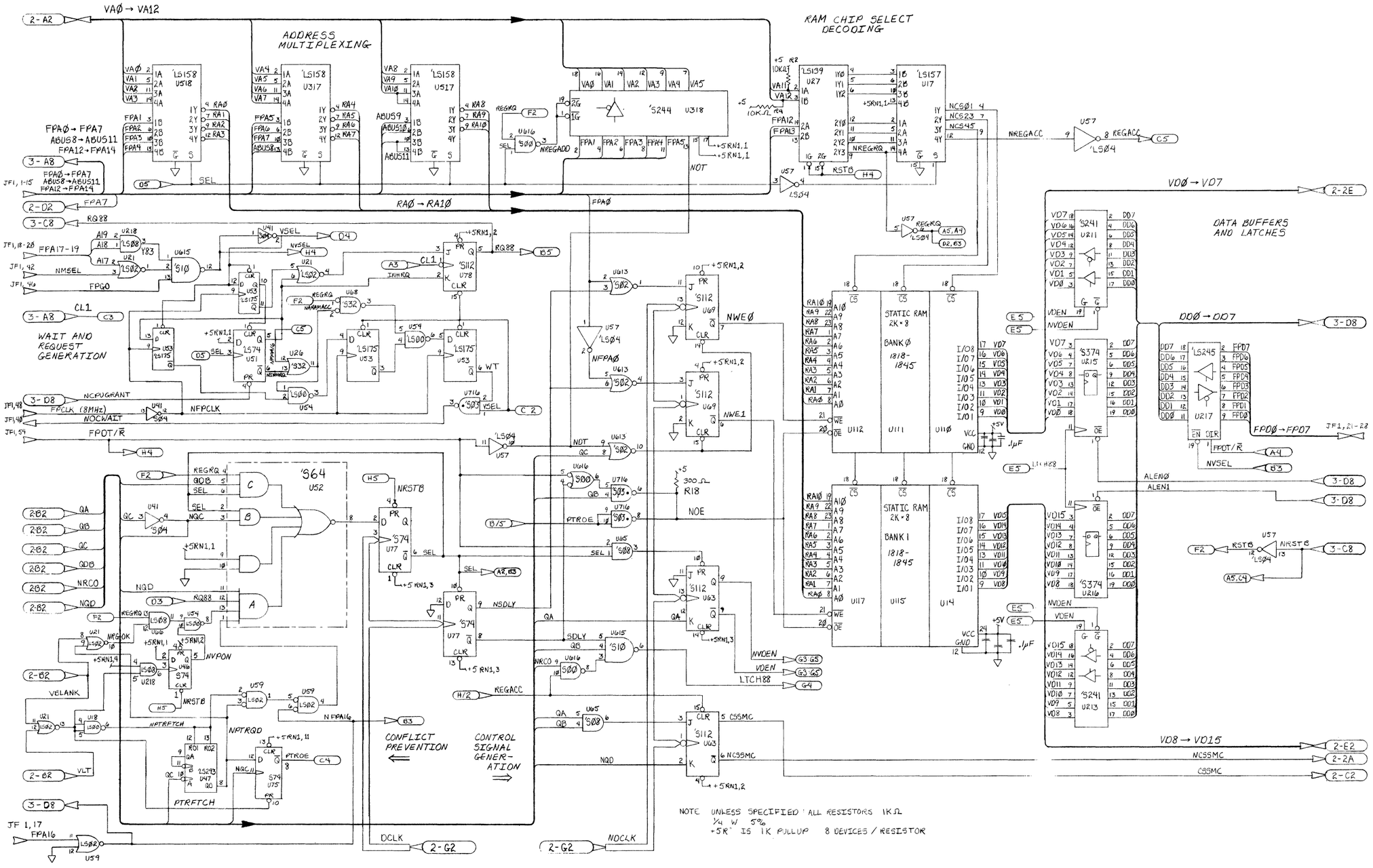


Figure A-7. Video Alpha RAM Subsystem

This page is blank

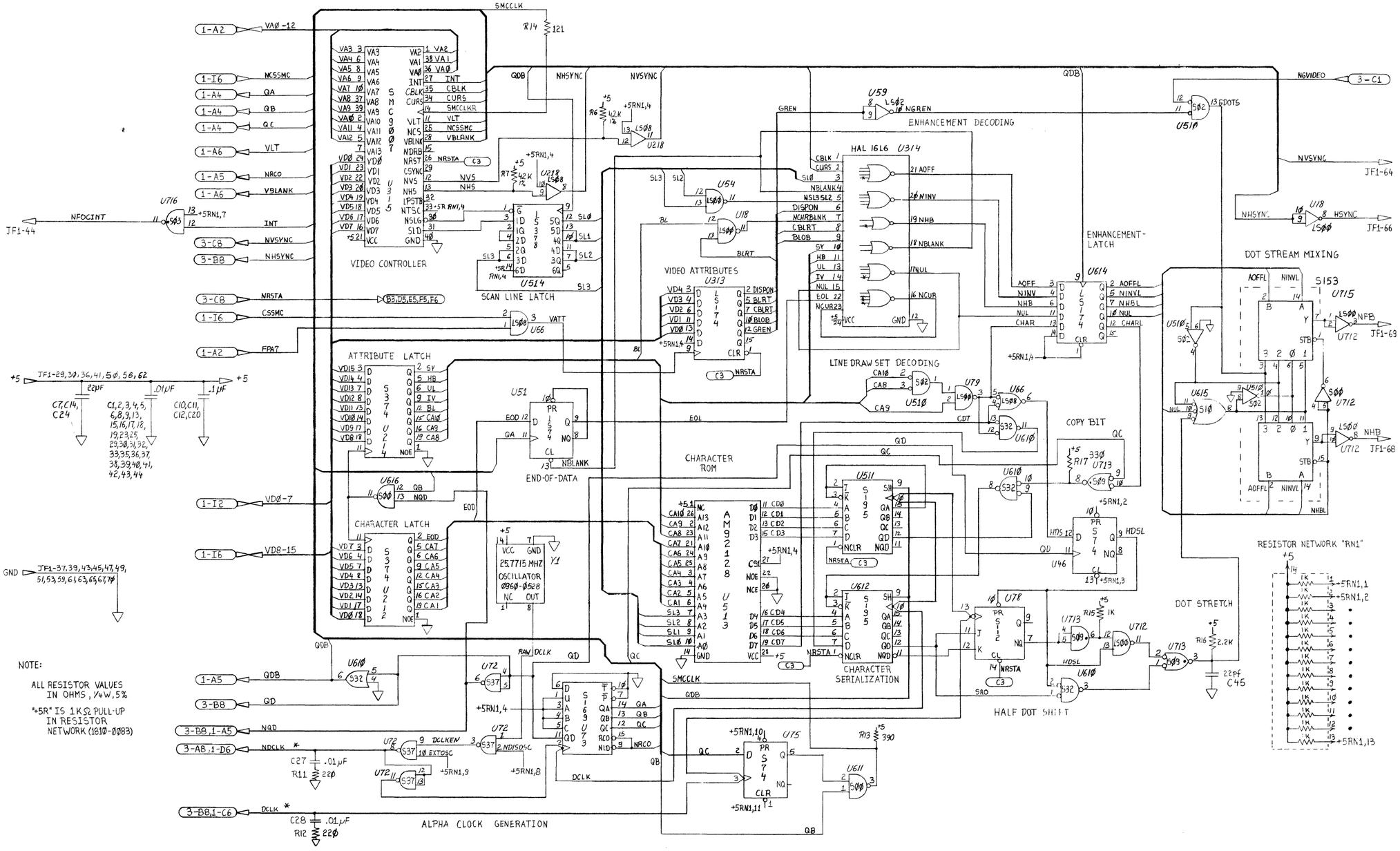


Figure A-8. Video Alpha Display Subsystem

This page is blank

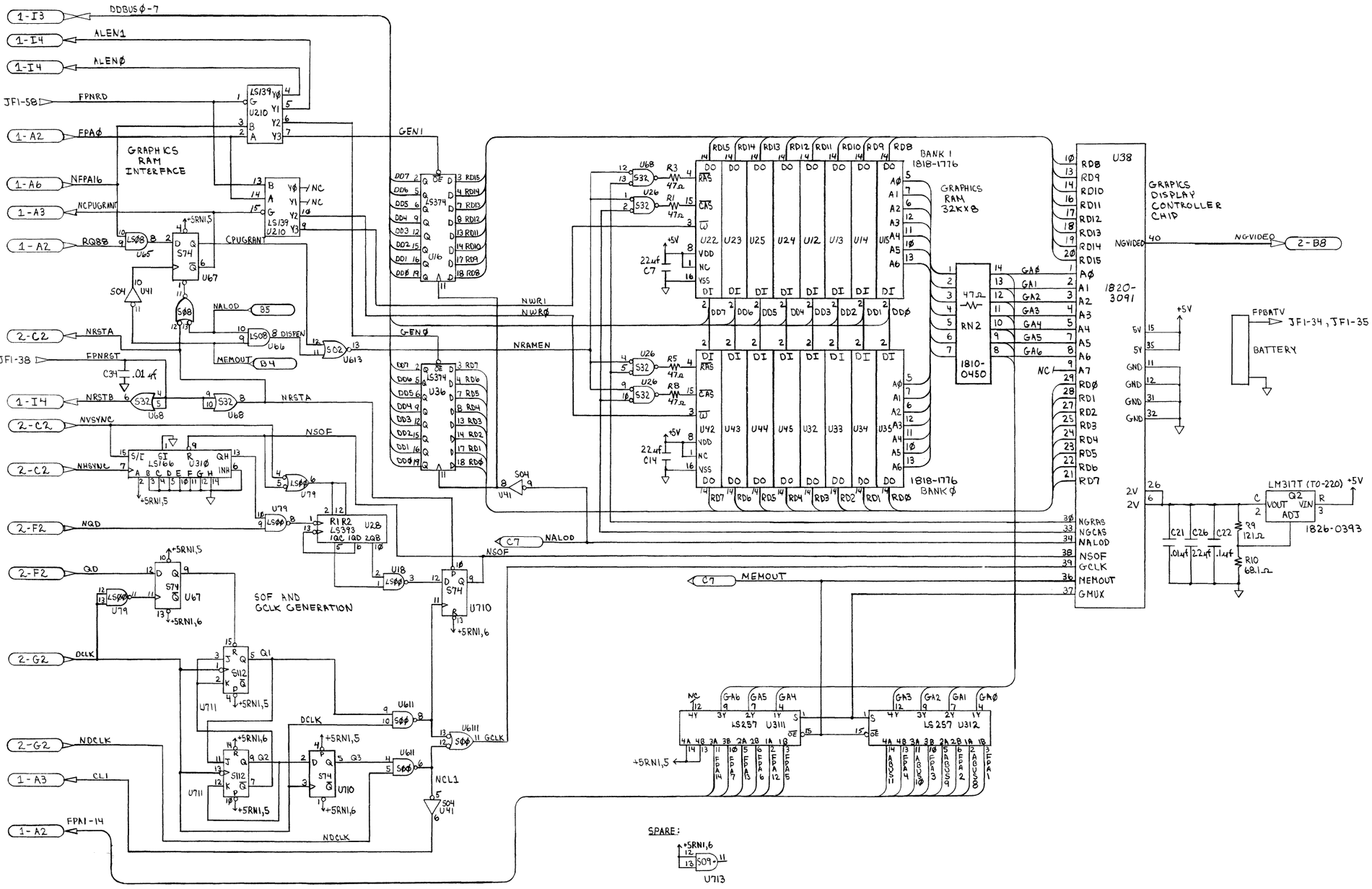


Figure A-9. Video Graphics Display Subsystem

This page is blank

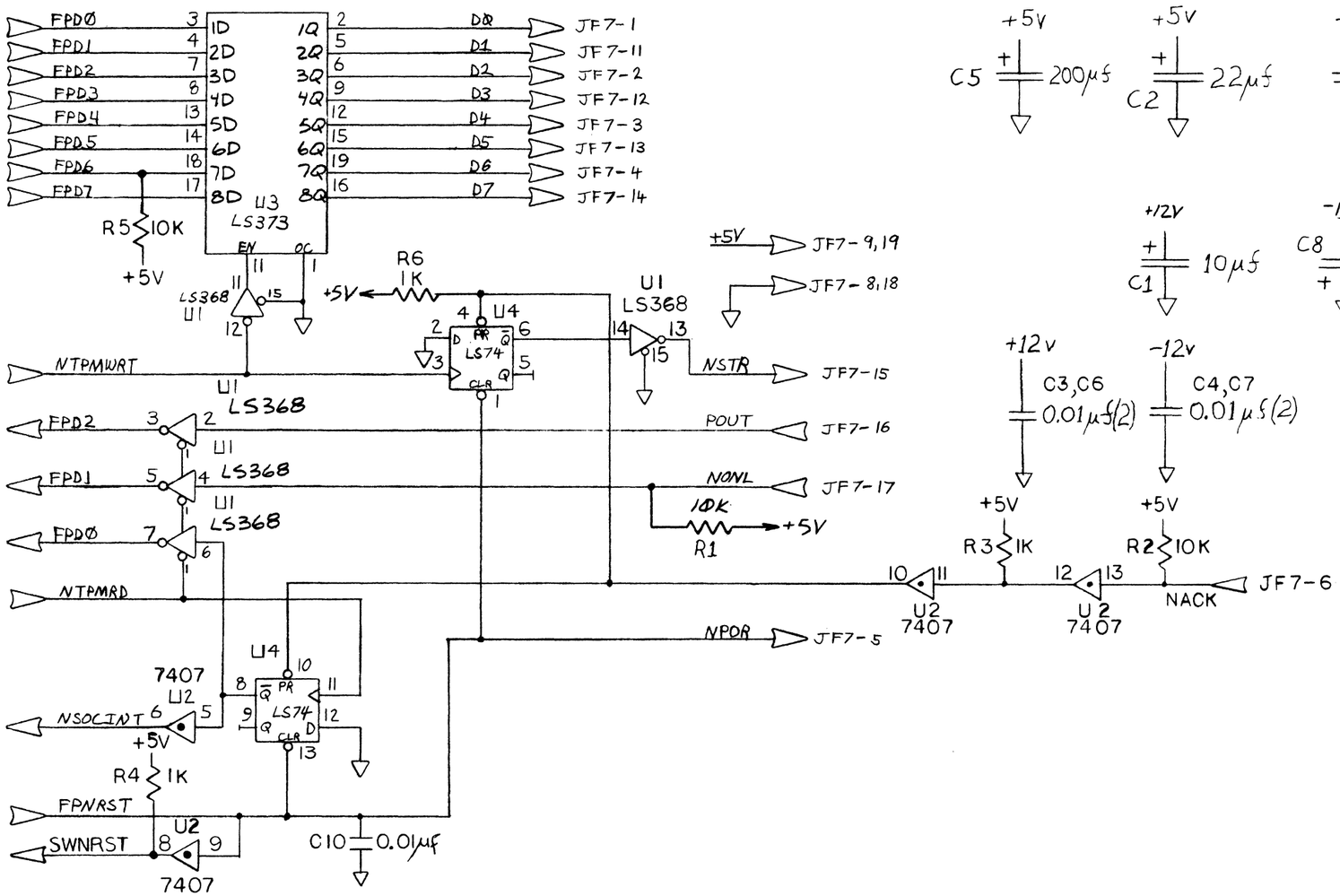


Figure A-10. Thermal Printer Interface (Part of Front Plane PCA)

This page is blank

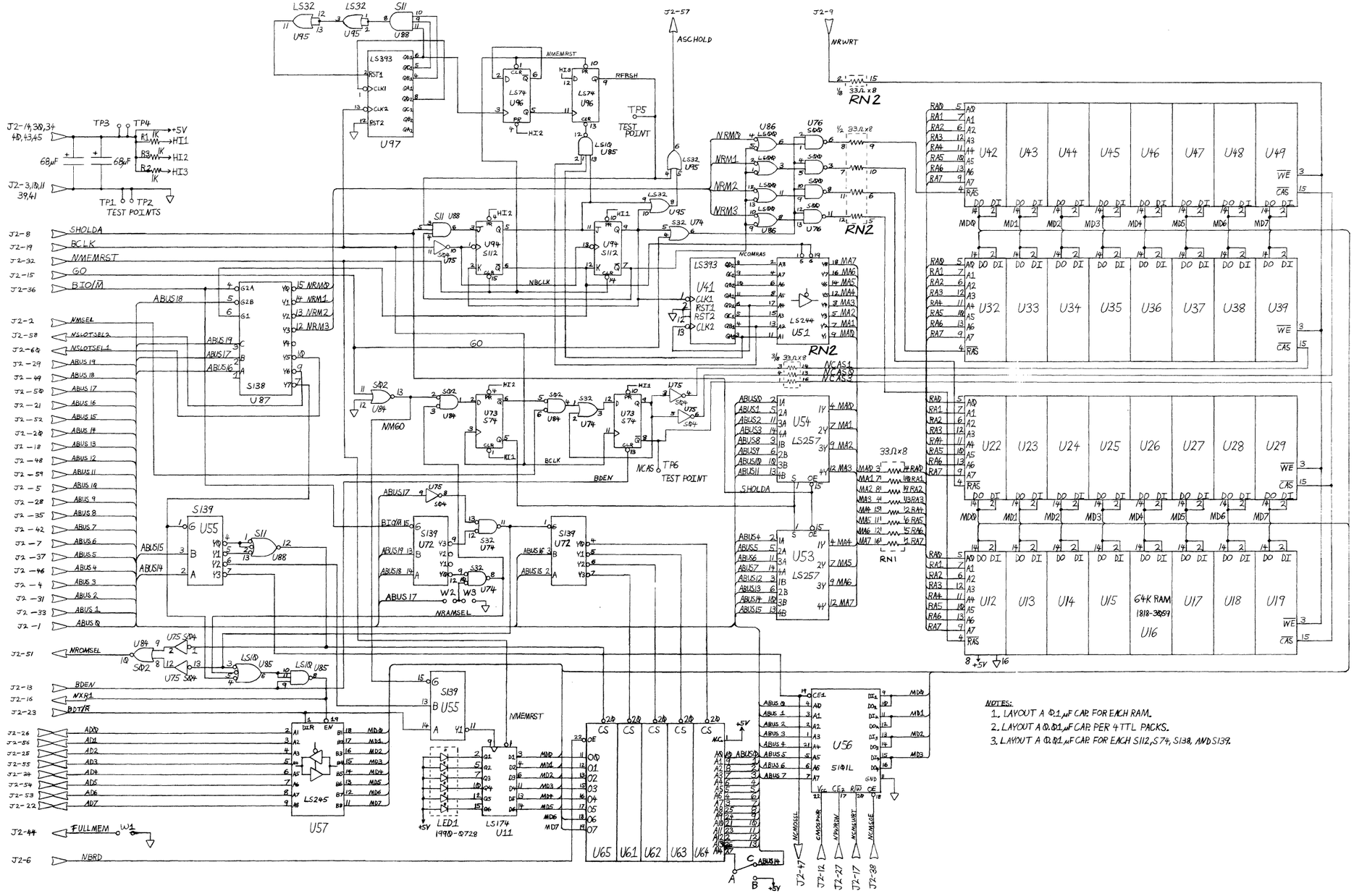
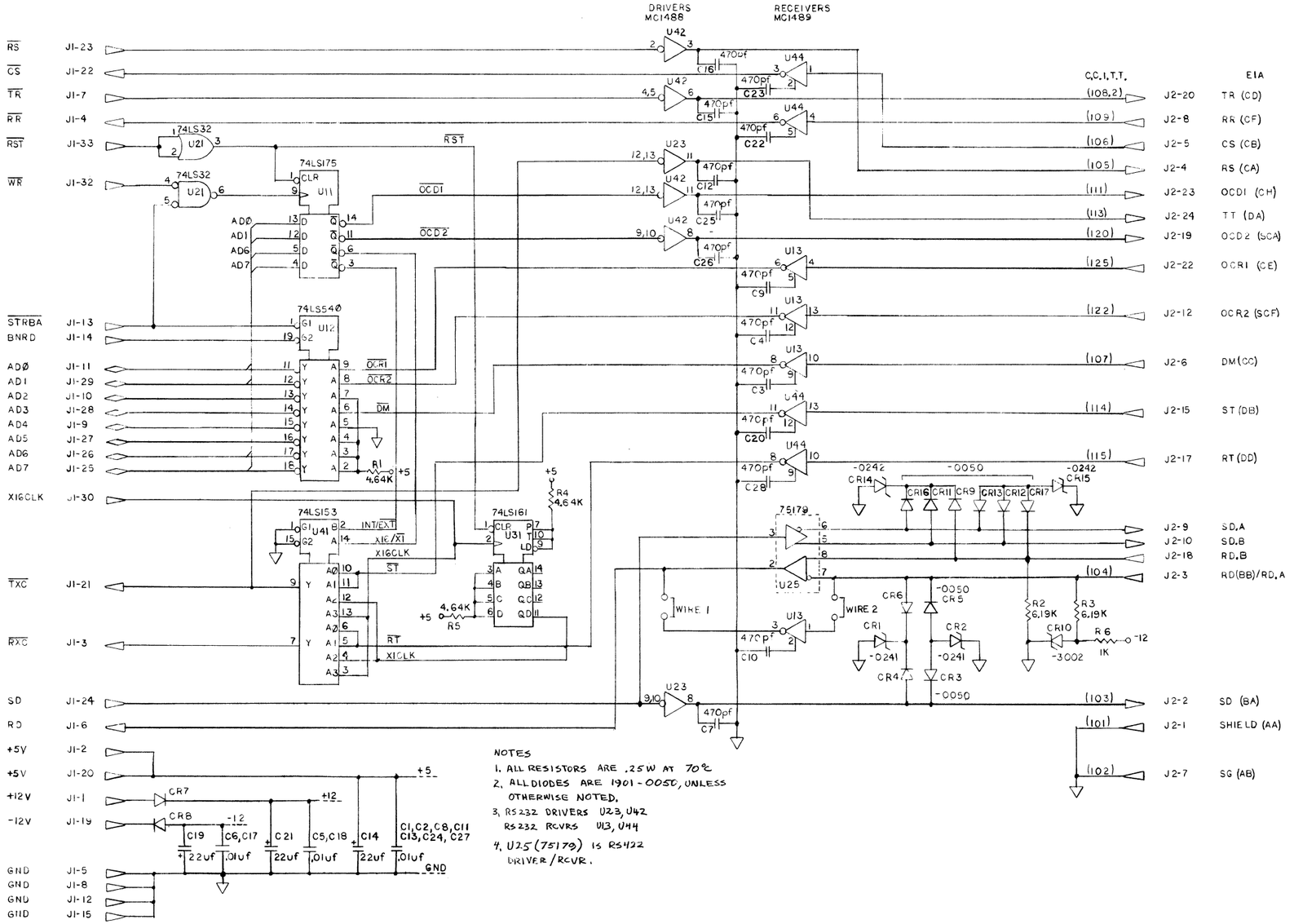


Figure A-11. Mezzanine Memory PCA

This page is blank



NOTES
1. ALL RESISTORS ARE .25W AT 70°C
2. ALL DIODES ARE 1901-0050, UNLESS OTHERWISE NOTED.
3. RS232 DRIVERS U23, U42
   RS232 RCVRS U13, U44
4. U25 (75179) IS RS422 DRIVER/RCVR.

Figure A-12. Mezzanine Datacomm PCA

This page is blank

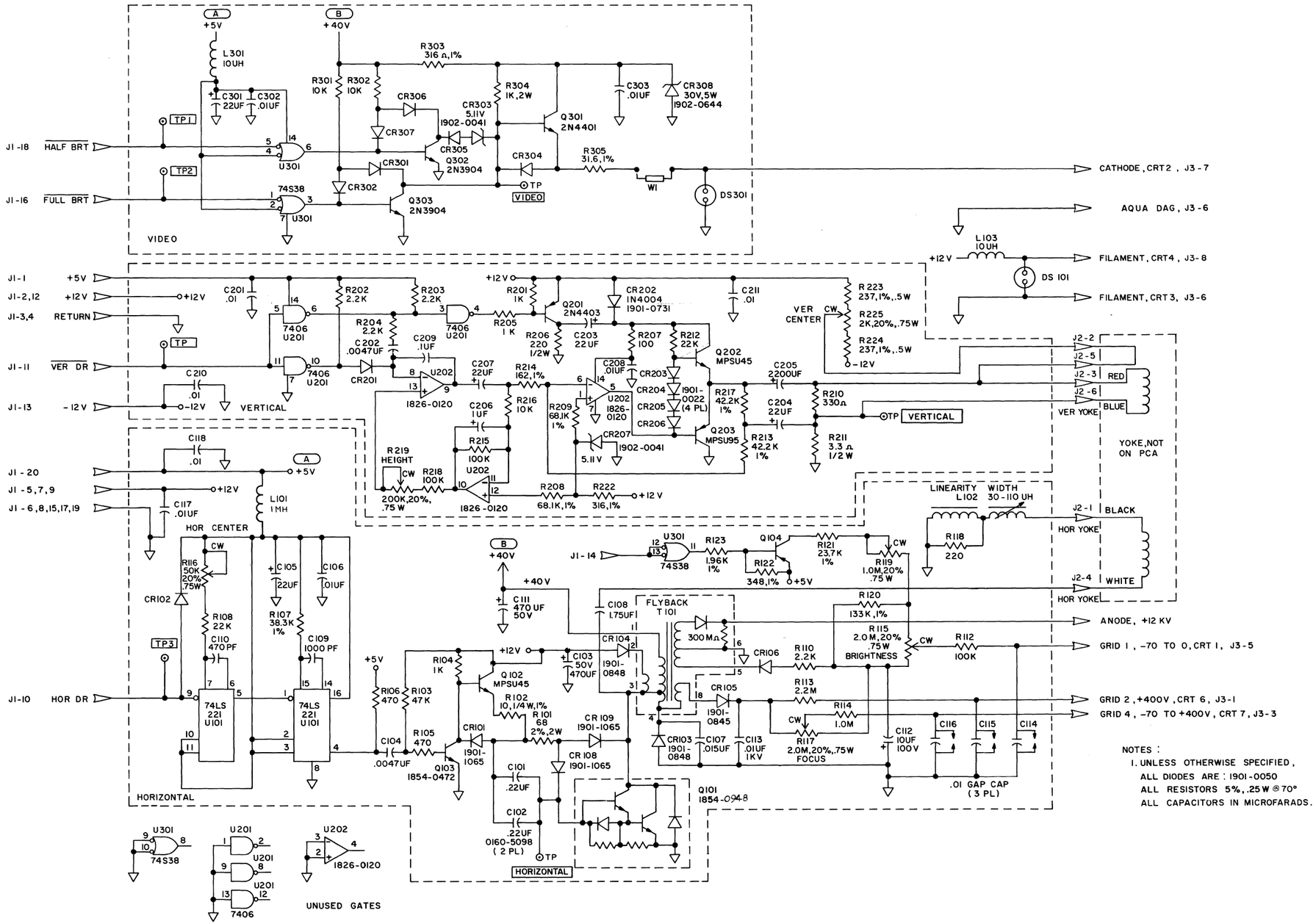


Figure A-13. Sweep PCA

This page is blank

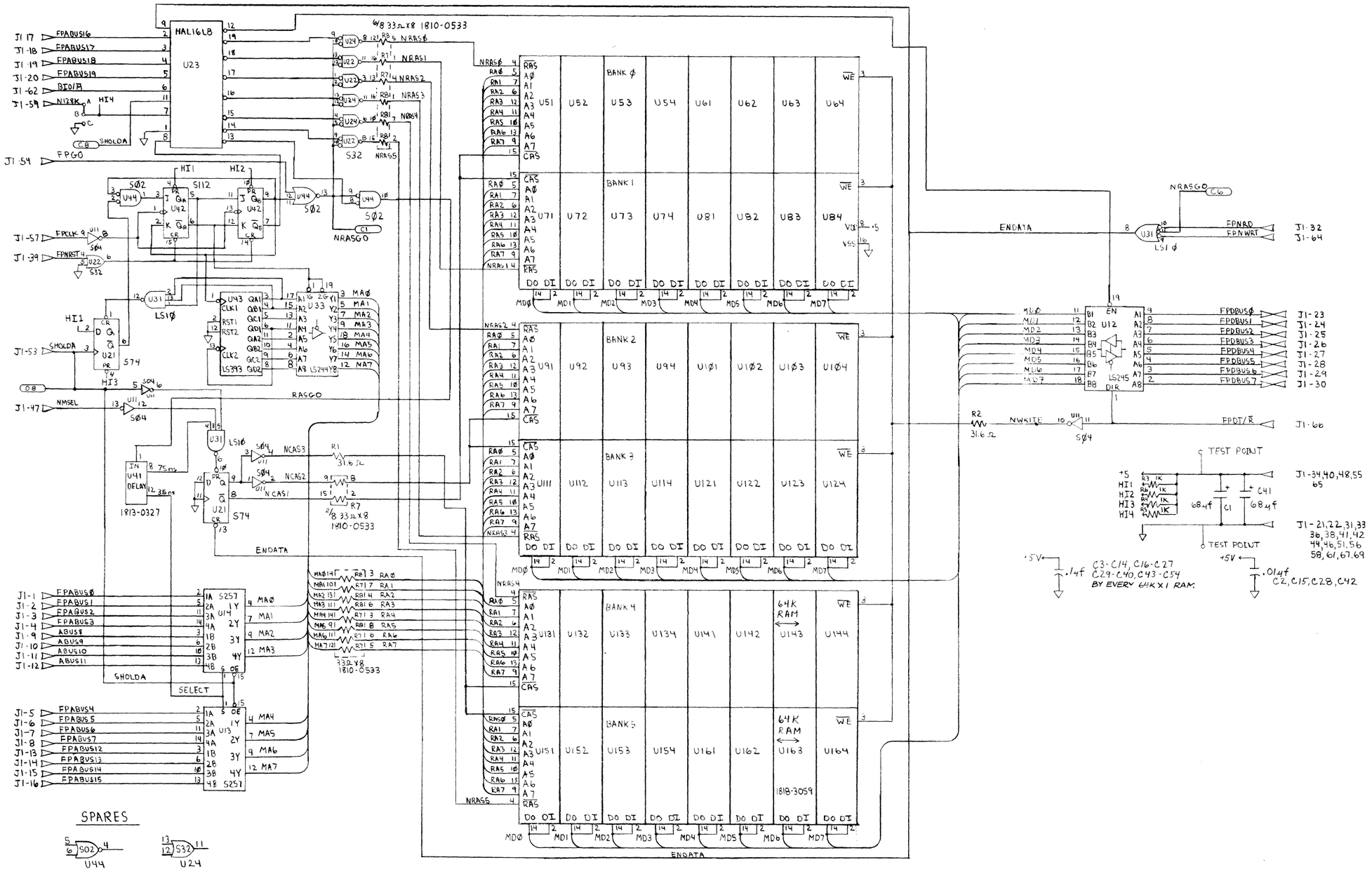


Figure A-14. RAM (Memory Extender) PCA

This page is blank

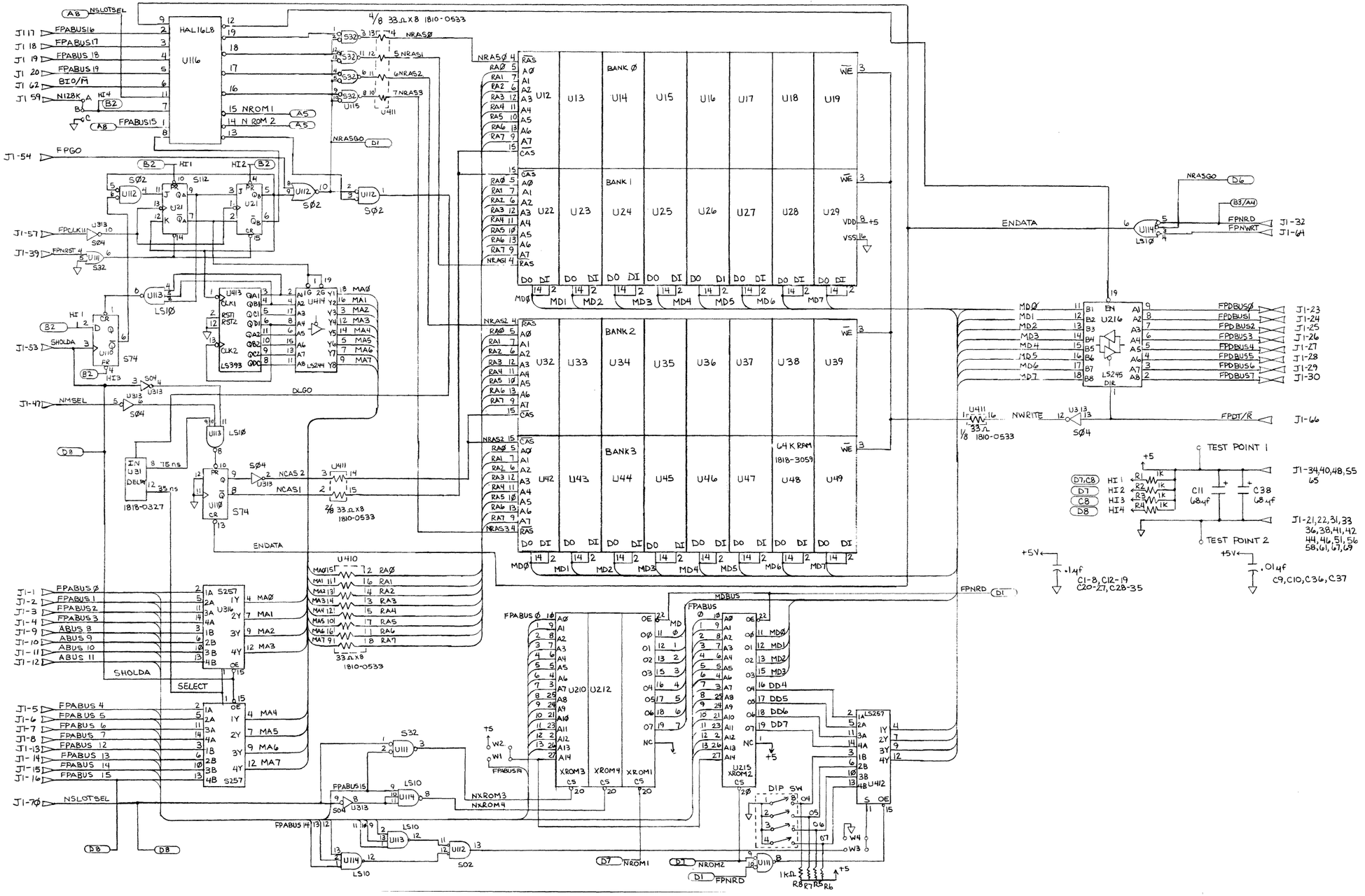


Figure A-15. Language PCA

FURTHER REFERENCE DOCUMENTS

APPENDIX

B

Additional information may be found in the following documents:

Title	HP Product No.	HP Part No.
o *HP 150 Service Manual*	45611A	45611-90002
o *HP 150 Owner's Guide*	45621A	45621-90001
o *HP 150 Terminal User's Guide*	45623A	45623-90002
o *MS-DOS User's Guide*	45624A	45624-90001

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y

READER COMMENT SHEET

HP 150 Technical Reference Manual

45625-90001 May 1984

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please use additional pages if necessary.

Is this manual technically accurate?	Yes [ ] No [ ]	(If no, explain under Comments, below.)
Are the concepts and wording easy to understand?	Yes [ ] No [ ]	(If no, explain under Comments, below.)
Is the format of this manual convenient in size, arrangement and readability?	Yes [ ] No [ ]	(If no, explain under Comments, below.)

Comments:

Date:_____

FROM:

Name_____ **Title**_____

Company_____

Address_____

City_____ **State**_____ **Zip**_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1355 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company
Personal Office Computer Division
P.O. Box 486
974 E. Arques Avenue
Sunnyvale, CA 94086

Attn: Technical Support Manager

FOLD FOLD

HEWLETT
PACKARD

Part No. 45625-90001
Printed in U.S.A. 5/84